Tweaks to the Kona Variant

Document #:	D0080R1
Date:	2015-10-20
Project:	Programming Language C++ Library Evolution Group

Reply-to:	Michael Park <mcypark@gmail.com>

1 Introduction

This paper proposes tweaks to the Kona `variant`.

2 Motivation and Scope

The main motivation is to increase consistency, usability and readability of the Kona `variant`. It does not question the fundamental design of `variant` which reached consensus, but mainly around the visitation mechanism.

3 Impact on the Standard

This proposal is a pure library extension, and does not require any new language features.

4 Proposed Wording

Make the following changes to `<experimental/optional>` header:

```
namespace std {
namespace experimental {
- constexpr in_place_t in_place{};
+ in_place_t in_place() { return {}; }

- template <class... Args> constexpr explicit optional(in_place_t, Args&&...);
+ template <class... Args> constexpr explicit optional(in_place_t (&)(), Args&&...);

  template <class U, class... Args>
- constexpr explicit optional(in_place_t, initializer_list<U>, Args&&...);
+ constexpr explicit optional(in_place_t (&)(), initializer_list<U>, Args&&...);
} // namespace experimental
} // namespace std
```

Make the following changes to `<experimental/variant>` header:

```
namespace std {
namespace experimental {
- template <class T> struct emplaced_type_t{};
- template <class T> constexpr emplaced_type_t<T> emplaced_type;
+ template <class T> in_place_t in_place(T) { return {}; }

- template <size_t I> struct emplaced_index_t{};
- template <size_t I> constexpr emplaced_index_t<I> emplaced_index;
+ template <size_t I> in_place_t in_place(integral_constant<size_t, I>) { return {}; }

  template <class... Types>
  class variant {
  public:
    /* ... */

    template <class T, class... Args>
-   constexpr explicit variant(emplaced_type_t<T>, Args&&...);
+   constexpr explicit variant(in_place_t (&)(T), Args&&...);

    template <class T, class U, class... Args>
-   constexpr explicit variant(emplaced_type_t<T>, initializer_list<U>, Args&&...);
+   constexpr explicit variant(in_place_t (&)(T), initializer_list<U>, Args&&...);

    template <size_t I, class... Args>
-   constexpr explicit variant(emplaced_index_t<I>, Args&&...);
+   constexpr explicit variant(in_place_t (&)(integral_constant<size_t, I>), Args&&...);

    template <size_t I, class U, class... Args>
-   constexpr explicit variant(emplaced_index_t<I>, initializer_list<U>, Args&&...);
+   constexpr explicit variant(in_place_t (&)(integral_constant<size_t, I>), initializer_list<U>, Args&&...);

    /* ... */
  };

- template <class Visitor, class... Variants>
- decltype(auto) visit(Visitor&&, Variants&&...);

+ template <class R = deduce_tag, class... Variants>
+ unspecified:TypeSwitch<R, Variants...> type_switch(Variants&&...);
} // namespace experimental
} // namespace std
```

Visitation [variant.visit]

```
template <class R = deduce_tag, class... Variants>
unspecified:TypeSwitch<R, Variants&&...> type_switch(Variants&&... vs);
```

> *Effects*: Constructs a callable object `TypeSwitch` which holds references to `forward<Variants>(vs)...`. Because the result may contain references to temporary variables, a program shall ensure that the return value of this function does not outlive any of its arguments. (e.g., the program should typically not store the result in a named variable).

```
template <class R, class... Variants>
struct TypeSwitch {
  explicit TypeSwitch(Variants...);

  template <class F>
  see below TypeSwitch::operator()(F&&) const;

  template <class... Fs>
  see below TypeSwitch::operator()(Fs&&...) const;

  template <template <class...> class F, class... Args>
  see below TypeSwitch::operator()(const typed_visitor<F, Args...>&) const;
};

explicit TypeSwitch(Variants... vs);
```

Effects: Holds references to `vs...`.

```
template <class F>
see below TypeSwitch::operator()(F&& f) const;
```

Let `vs...` be the references to instances of `Variants...` that *`TypeSwitch`* holds.

Let $\texttt{Variant}_\texttt{I}$ be the $\texttt{I}^{\texttt{th}}$ type in `decay_t<Variants>>...`.

Let $\texttt{Types}_\texttt{I}\texttt{...}$ be the template parameter pack `Types...` of $\texttt{Variant}_\texttt{I}$.

Requires: `forward<F>(f).template operator()<`$\texttt{I}_0$`, ..., `$\texttt{I}_N$`>(get<`$\texttt{I}_0$`>(`$\texttt{v}_0$`), ..., get<`$\texttt{I}_N$`>(`$\texttt{v}_N$`))` `OR invoke(forward<F>(f), get<`$\texttt{I}_0$`>(`$\texttt{v}_0$`), ..., get<`$\texttt{I}_N$`>(`$\texttt{v}_N$`))` must be a valid expression for all $\texttt{I}_\texttt{I}$ where $\texttt{I}_\texttt{I} \in$ `[ 0, sizeof...(`$\texttt{Types}_\texttt{I}$`)]`.

Effects: Let `Is...` be the constant expression of `vs.index()...`. Equivalent to `forward<F>(f).template operator()<Is...>(get<Is>(vs)...)` if it is a valid expression, otherwise equivalent to `invoke(forward<F>(f), get<Is>(vs)...)`;

Remarks: If `!is_same_v<R, `*`unspecified`*`:deduce_tag>` is `true`, the return type of this function is `R`.

Otherwise, if `decay_t<F>` contains a member typedef `result_type`, the return type is `typename decay_t<F>::result_type`.

Otherwise, the return type is deduced by `decltype(auto)`.

```
template <class... Fs>
see below TypeSwitch::operator()(Fs&&... fs) const;
```

Effects: Equivalent to `(*this)(overload(forward<Fs>(fs)...)); // P0051`

5 Design Discussion

5.1 `Motifications to visit`

[N3915] proposed `apply` which takes a function object `F` and a tuple `Tuple`, and dispatches `F` with the elements of `Tuple`. The `visit` function was introduced to accompany the Kona `variant`. This function is similar to `apply`, as it takes a function object `F` and `Variants...`, and dispatches `F` with the content of each the `Variants...`.

Although the similarity to `apply` may seem desirable for consistency, I claim that there are major drawbacks at the callsite. The inherent difference between `apply` and `visit` is that `apply` only requires a single handler that handles the elements of the `tuple`, whereas `visit` (almost always) requires many handlers that handle the cartesian product of the alternatives of the `variant` objects.

Consider a few sample usages of visiting a `variant<int, std::string>` The following are 3 of the possible ways to provide the handlers via the `visit` interface.

- Function object which requires out-of-line definition.
- Generic lambda to provide a single generic handler inline.
- `overload` to provide specific handlers inline. // P0051

```
struct Print {
  void operator()(int value) const { std::cout << "int: " << value; }
  void operator()(const std::string& value) const { std::cout << "string: " << value; }
};

int main() {
  std::variant<int, std::string> v(42);

  // Function object defined out-of-line.
  std::visit(Print{}, v);

  // Generic lambda to provide a generic handler inline.
  std::visit([](const auto& value) { std::cout << value; }, v);

  // ‘overload‘ to provide specific handlers inline.  // P0051
  std::visit(
    overload([](int value) { std::cout << "int: " << value; },
             [](const std::string& value) { std::cout << "string: " << value; }),
    v);

  // copy construction.
  std::variant<int, std::string> w(v);

  // multi-visitation
  std::visit(
      overload([](int, int) { std::cout << "(int, int)"; },
               [](int, const std::string &) { std::cout << "(int, string)"; },
               [](const std::string &, int) { std::cout << "(string, int)"; },
               [](const std::string &, const std::string &) { std::cout << "(string, string)"; }),
      v, w);
}
```

Aside from `overload` not yet existing in the standard as a utility, it also flips the order of variants and handlers. It would be enhance readability to list the variants to be handled, followed by the handlers (similar to the structure of `switch`).

Constrast this with `type_switch`.

```
struct Print {
  void operator()(int value) const { std::cout << "int: " << value; }
  void operator()(const std::string& value) const { std::cout << "string: " << value; }
};

int main() {
  std::variant<int, std::string> v(42);

  // Function object defined out-of-line.
  std::type_switch (v) (Print{});

  // Generic lambda to provide a generic handler inline.
  std::type_switch (v) (
    [](const auto& value) { std::cout << value; }
  );

  // No need for ‘overload‘.
  std::type_switch (v) (
    [](int value) { std::cout << "int: " << value; },
    [](const std::string& value) { std::cout << "string: " << value; }
  );

  // copy construction.
  std::variant<int, std::string> w(v);

  // multi-visitation.
  std::type_switch (v, w) (
    [](int               , int               ) { std::cout << "(int, int)"      ; },
    [](int               , const std::string&) { std::cout << "(int, string)"   ; },
    [](const std::string&, int               ) { std::cout << "(string, int)"   ; },
    [](const std::string&, const std::string&) { std::cout << "(string, string)"; }
  );
}
```

5.2 Explicit Return Type

In order to mitigate the requirement that all handlers must be the same type, it would be convenient to allow the user to explicitly provide the return type.

Rather than modifying every handler to return the same type, the user can explicitly provide the return type to which all return types of the handlers must be implicitly convertible to.

```
std::variant<int, std::string> v(42);
auto result = std::type_switch<int64_t> (v) (
  [](int value) { return value; },
  [](const std::string& value) { return value.size(); }
);
// decltype(result) == int64_t
```

5.3 Index-Aware Handlers

In cases where it is useful to know the original type in which the value came from, an index-aware handler can be provided to retrieve the indices of the originating types. This situation can arise from duplicate types, but also from the consequence of allowing references.

```
int x = 42;
variant<int, int&> v(in_place<1>, x);
type_switch (v) ( [](int&) { /* matches both int, and int& */ );

// index-aware handler
struct index_aware {
  template <size_t I>
  void operator()(int&) const {
    /* still matches both int, and int&, but 'I' tells you which one. */
  }
};

type_switch (v) ( index_aware{} );
```

Since non-deduced template parameters cannot be specified in a lambda, the implication here is that an index aware handler must be defined as an out-of-line function object. However, considering that this is likely a rare use case, it is unlikely to be an issue.

6 Acknowledgements

- **Eric Friedman** and **Itay Maman** for the development of [Boost.Variant].
- **Agustín Bergé** (a.k.a K-ballo) for the development of [Eggs.Variant].
- **Axel Naumann** for writing [N4542] and its predecessors which started the discussion.
- **David Sankel** for deep design discussions and encouraging me to present my ideas during CppCon 2015.
- **Kirk Shoop** for design discussions at CppCon 2015.
- **Eric Niebler** for letting me know that submitting a proposal is a must in order to be considered for standardization.
- **Geoffrey Romer** for encouraging me to write this paper since CppCon 2014.
- **Andrei Alexandrescu** for starting the work on `variant`, writing Dr. Dobb's articles back in 2002, and for encouraging me to write this paper.
- Everyone on **std-proposals** mailing list who passionately share their thoughts, ideas, and opinions.

A special thank you to **Jason Lucas** for initially introducing me to the multi-method problem, and for the endless design and tradeoff discussions over the past few years!

7 References

[N3449] Bjarne Stroustrup, *Open and Efficient Type Switch for C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3449.pdf

[N3915] Peter Sommerlad, *apply() call a function with arguments from a tuple (V3)*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3915.pdf

[N4542] Axel Naumann, *Variant: a type-safe union (v4)*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4542.pdf

[P0050] Vicente J. Botet Escriba, *C++ generic match function*
https://github.com/viboes/tags/blob/master/doc/proposals/match/P0050.pdf

[P0051] Vicente J. Botet Escriba, *C++ generic overload functions*
https://github.com/viboes/tags/blob/master/doc/proposals/overload/P0051.pdf

[The Billion Dollar Mistake] Tony Hoare, *Null References: The Billion Dollar Mistake*
http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

[Mach7] Yuriy Solodkyy, Gabriel Dos Reis, Bjarne Stroustrup, *Mach7: Pattern Matching for C++*
https://github.com/solodon4/Mach7

[Boost.Variant] Eric Friedman, Itay Maman, *Boost.Variant*
http://www.boost.org/doc/libs/1_59_0/doc/html/variant.html

[Eggs.Variant] Agustín Bergé, *Eggs.Variant*
http://eggs-cpp.github.io/variant

[MPark.Variant] Michael Park, *Variant: Discriminated Union with Value Semantics*
https://github.com/mpark/variant

[C++ Core Guidelines] Bjarne Stroustrup, Herb Sutter *C++ Core Guidelines*
https://github.com/isocpp/CppCoreGuidelines