Variant: a typesafe union. ISO/IEC JTC1 SC22 WG21 N4218

Axel Naumann (axel@cern.ch)

2014-09-24

Contents

Introduction

C++ needs a typesafe union; here is a proposal. It attempts to apply the lessons learned from `optional` (1). It behaves as below:

```
variant<int, float> v, w;
v = 12;
int i = get<int>(v);
w = get<int>(v);
w = get<0>(v); // same effect as the previous line
w = v; // same effect as the previous line

get<double>(v); // ill formed
get<3>(v); // ill formed
```

```
try {
  get<float>(w); // will throw.
}
catch (bad_variant_access&) {}
```

Discussion

A `variant` is not `boost::any`

A `variant` stores one value out of multiple possible types (the template parameters to `variant`). It can be seen as a restriction of `any`. Given that the types are known at compile time, `variant` allows the storage of the value to be contained inside the `variant` object.

`union` versus `variant`

This propsal is not meant to replace `union`: its undefined behavior when casting `Apples` to `Oranges` is an often used feature that distinguishes it from `variant`'s features. So be it.

On the other hand, `variant` is able to store values with non-trivial constructors and destructors. Part of its visible state is the type of the value it holds at a given moment; it enforces value access happening only to that type.

Other implementations

The C++ `union` is a non-typesafe version of `variant`. `boost::variant` (2) is very similar to this proposal. This proposal tries to merge the lessons from `optional`. It does not address the visitation pattern that is an integral part of the `boost::variant`, nor any special treatment of a recursive `variant`.

Design considerations

A `variant` can be empty

To simplify the `variant` it should always contain a value of one of its template type parameters. But this would have a major drawback:

In the state transition of an assignment of two `variants` `v`, `w` of same type:

```
variant<S, T> v = S();
variant<S, T> w = T();
v = w;
```

`v` will first destruct its current value of type `S`, then initialize the new value from the value of type `T` that is held in `w`. If the latter part fails (for instance throwing an exception), `v` will not contain any valid value. It must not destruct the contained value as part of `~variant`, and it must make this state visible, because any call of `get<T>(v)` would access an invalid object. The most straight-forward option is to introduce a new, empty state of the `variant`.

C++ `union`s get around this by not allowing type transitions. We find assignments involving type transitions too desirable to forbid them.

Another advantage for the empty state is unambiguous behavior of the default constructor.

Variant Objects

In general

Variant objects contain and manage the lifetime of a value. If the variant is not empty, the single contained value's type has to be one of the template argument types given to `variant`. These template arguments are called alternatives.

Header `<variant>` synopsis

```
namespace std {
namespace experimental {
inline namespace fundamentals_vXXXX {
  // 2.?, variant of value types
  template <class... Types> class variant;

  // 2.?, In-place construction
  struct in_place_t{};
  constexpr in_place_t in_place{};

  // 2.?, variant creation functions
  // ==> Cannot think of anything reasonable here

  // 2.?, class bad_variant_access
  class bad_variant_access;

  // 2.?, variant helper classes
  template <class T> class variant_num_alternatives;  // undefined
  template <class T> class variant_num_alternatives<const T>;
  template <class T> class variant_num_alternatives<volatile T>;
  template <class T> class variant_num_alternatives<const volatile T>;
```

```
template <class... Types>
  class variant_num_alternatives<variant<Types...> >;

template <size_t I, class T>
  class variant_alternative;    // undefined
template <size_t I, class T>
  class variant_alternative<I, const T>;
template <size_t I, class T>
  class variant_alternative<I, volatile T>;
template <size_t I, class T>
  class variant_alternative<I, const volatile T>;

template <size_t I, class... Types>
  class variant_alternative<I, variant<Types...> >;

template <size_t I, class T>
  using variant_alternative_t
    = typename variant_alternative<I, T>::type;

// 2.?, handling of alternatives
template <class T, class... Types>
  constexpr bool is_alternative(const variant<Types...>&) noexcept;
static const size_t alternative_unavailable = (size_t) -1;
template <class T, class... Types>
  constexpr size_t alternative_index(const variant<Types...>&) noexcept;

// 2.?, value access
template <class T, class... Types>
  bool holds_alternative(const variant<Types...>&) noexcept;
template <size_t I, class... Types>
  variant_alternative_t<I, variant<Types...>>&
    get(variant<Types...>&);
template <size_t I, class... Types>
  variant_alternative_t<I, variant<Types...>>&&
    get(variant<Types...>&&);
template <size_t I, class... Types>
  const variant_alternative_t<I, variant<Types...>>&
    get(const variant<Types...>&);
template <class T, class... Types>
  T& get(variant<Types...>&);
template <class T, class... Types>
  T&& get(variant<Types...>&&);
template <class T, class... Types>
  const T& get(const variant<Types...>&);
```

```
  // 2.?, relational operators
  template <class... TTypes, class... UTypes>
    bool operator==(const variant<TTypes...>&,
                    const variant<UTypes...>&);
  template <class... TTypes, class... UTypes>
    bool operator!=(const variant<TTypes...>&,
                    const variant<UTypes...>&);
  template <class... TTypes, class... UTypes>
    bool operator<(const variant<TTypes...>&,
                    const variant<UTypes...>&);
  template <class... TTypes, class... UTypes>
    bool operator>(const variant<TTypes...>&,
                    const variant<UTypes...>&);
  template <class... TTypes, class... UTypes>
    bool operator<=(const variant<TTypes...>&,
                    const variant<UTypes...>&);
  template <class... TTypes, class... UTypes>
    bool operator>=(const variant<TTypes...>&,
                    const variant<UTypes...>&);

  // 2.?, Specialized algorithms
  template <class... Types>
    void swap(variant<Types...>& x, variant<Types...>& y);

} // namespace fundamentals_vXXXX
} // namespace experimental

  // 2.?, Hash support
  template <class... Types> struct hash;
  template <class... Types>
    struct hash<experimental::variant<Types...>>;
} // namespace std
```

Class template `variant`

```
namespace std {
namespace experimental {
inline namespace fundamentals_vXXXX {
  template <class... Types>
  class variant {
    // 2.? variant construction
    constexpr variant() noexcept;
    variant(const variant&);
    variant(variant&&) noexcept(see below);
```

```
    template <class T> constexpr explicit variant(const T&);
    template <class T> constexpr explicit variant(T&&);

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

    // 2.?, Destructor
    ~variant();

    // allocator-extended constructors
    template <class Alloc>
      variant(allocator_arg_t, const Alloc& a);
    template <class Alloc, class T>
      variant(allocator_arg_t, const Alloc& a, T);
    template <class Alloc>
      variant(allocator_arg_t, const Alloc& a, const variant&);
    template <class Alloc>
      variant(allocator_arg_t, const Alloc& a, variant&&);

    // 2.?, `variant` assignment
    variant& operator=(const variant&);
    variant& operator=(variant&&) noexcept; //see below
    template <class T> variant& operator=(U&&);
    template <class T, class... Args> void emplace(Args&&...);
    template <class T, class U, class... Args>
      void emplace(initializer_list<U>, Args&&...);
    void clear();

    // 2.?, value status
    bool empty() const nothrow;
    size_t index() const;

    // 2.?, `variant` swap
    void swap(variant&) noexcept; //see below

  private:
    static const size_t max_alternative_sizeof = ...; // exposition only
    char storage[max_alternative_sizeof]; // exposition only
    size_t value_type_index; // (size_t)-1 if empty; exposition only
  };
} // namespace fundamentals_vXXXX
} // namespace experimental
```

```
} // namespace std
```

Construction

For each `variant` constructor, an exception is thrown only if the construction of one of the types in `Types` throws an exception.

The defaulted move and copy constructor, respectively, of `variant` shall be a `constexpr` function if and only if all required element-wise initializations for copy and move, respectively, would satisfy the requirements for a `constexpr` function. The defaulted move and copy constructor of `variant<>` shall be `constexpr` functions.

In the descriptions that follow, let `i` be in the range `[0,sizeof...(Types))` in order, and `T_i` be the `i`$^{\text{th}}$ type in `Types`.

`constexpr variant() noexcept`

Effects: Constructs an empty `variant`.
Postconditions: `empty()` is `true`.

`variant(const variant& w)`

Requires: `is_copy_constructible<T_i>::value` is `true` for all `i`.
Effects: initializes the `variant` to hold the same alternative as `w`, or to an empty state if `w` was empty. If non-empty, initializes the contained value to a copy of the value contained by `w`.
Throws: Any exception thrown by the selected constructor of any `T_i` for all `i`.

`variant(variant&& w) noexcept(see below)`

Requires: `is_move_constructible<T_i>::value` is `true` for all `i`.
Effects: initializes the `variant` to hold the same alternative as `w`. Initializes the contained value with `std::forward<T_j>(get<j>(w))` with `j` being `w.index()`.
Postconditions: `empty() && w.empty() || holds_alternative<T>(*this) == holds_alternative<T>(w)` is `true`.
Throws: Any exception thrown by the selected constructor of any `T_i` for all `i`.
Remarks: The expression inside `noexcept` is equivalent to the logical AND of `is_nothrow_move_constructible<T_i>::value` for all `i`.

`template <class T> constexpr explicit variant(const T& t)`

Requires: `is_alternative<T>(variant())` is `true` and `is_copy_constructible<Y>::value` is `true`.

Effects: initializes the `variant` to hold the alternative `T`. Initializes the contained value to a copy of `t`.

Postconditions: `holds_alternative<T>(*this)` is `true`

Throws: Any exception thrown by the selected constructor of T.

Remarks: If `T`'s selected constructor is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

`template <class T> constexpr explicit variant(T&& t)`

Requires: `is_alternative<T>(variant())` is `true` and `is_move_constructible<T>::value` is `true`.

Effects: initializes the `variant` to hold the alternative `T`. Initializes the contained value with `std::forward<T>(t)`.

Postconditions: `holds_alternative<T>(*this)` is `true`

Throws: Any exception thrown by the selected constructor of T.

Remarks: If `T`'s selected constructor is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

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

Requires: `is_alternative<T>(variant())` is `true` and `is_constructible<T, Args&&...>::value` is `true`.

Effects: Initializes the contained value as if constructing an object of type `T` with the arguments `std::forward<Args>(args)...`.

Postcondition: `holds_alternative<T>(*this)` is `true`

Throws: Any exception thrown by the selected constructor of `T`.

Remarks: If `T`'s selected constructor is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

`template <class T, class U, class... Args> constexpr explicit variant(in_place_t, initializer_list<U> il, Args&&...);`

Requires: `is_alternative<T>(variant())` is `true` and `is_constructible<T, initializer_list<U>&, Args&&...>::value` is `true`.

Effects: Initializes the contained value as if constructing an object of type `T` with the arguments `il, std::forward<Args>(args)...`.

Postcondition: `holds_alternative<T>(*this)` is `true`

Remarks: The function shall not participate in overload resolution unless `is_constructible<T, initializer_list<U>&, Args&&...>::value` is `true`.

Remarks: If `T`'s selected constructor is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

Destructor

`~variant()`

Effects: If `empty()` is `false`, calls `get<T_j>(*this).T_j::~T_j()` with `j` being `this.index()`.

Assignment

`variant& operator=(const variant& rhs)`

Requires: `is_copy_constructible<T_i>::value` is `true` and `is_copy_assignable<T_i>::value` is `true` for all `i`.

Effects: destructs the current contained value of `*this` if `empty()` is `false`. Initializes `*this` to hold the same alternative as `rhs`. Initializes the contained value to a copy of the value contained by `rhs`.

Returns: `*this`.

Postconditions: `this->index() == rhs.index()`

Exception safety: If any exception is thrown by the destruction of the old value of `this`, `this->empty()` will be `true` and no copy assignment will take place. If `rhs.empty()` is false and an exception is thrown during the call to `T_j`'s copy constructor (with `j` being `rhs.index()`, `this->empty()` will be `true` and no copy assignment will take place. If `rhs.empty()` is false and an exception is thrown during the call to `T_i`'s copy assignment, the state of the contained value is as defined by the exception safety guarantee of `T_i`'s copy assignment; `this->index()` will be `j`.

`variant& operator=(const variant&& rhs) noexcept (see below)`

Requires: `is_move_constructible<T_i>::value` is `true` and `is_move_assignable<T_i>::value` is `true` for all `i`.

Effects: destructs the current contained value of `*this` if `empty()` is `false`. Initializes `*this` to hold the same alternative as `rhs`. Initializes the contained value with `std::forward<T_j>(get<j>(rhs))` with `j` being `w.index()`.

Returns: `*this`.
Remarks: The expression inside `noexcept` is equivalent to:

`is_nothrow_move_assignable<T_i>::value && is_nothrow_move_constructible<T_i>::value` for all `i`. Postconditions: : `this->index() == rhs.index()` Exception safety: : If any exception is thrown by the destruction of the old value of `this`, `this->empty()` will be `true` and no copy assignment will take place. If `rhs.empty()` is false and an exception is thrown during the call to `T_j`'s copy constructor (with `j` being `rhs.index()`), `this->empty()` will be `true` and no copy assignment will take place. If `rhs.empty()` is false and an exception is thrown during the call to `T_i`'s copy assignment, the state of the contained value is as defined by the exception safety guarantee of `T_i`'s copy assignment; `this->index()` will be `j`.

`template <class T> variant& operator=(T&& t)`

Requires: `is_move_constructible<T>::value` is `true` and `is_move_assignable<T>::value` is `true`.
Effects: destructs the current contained value of `*this`. Initializes `*this` to hold the same alternative as `rhs`. Initializes the contained value with `std::forward<T>(t)`.
Returns: `*this`.
Postcondition: `holds_alternative<T>(*this)` is `true`.
Exception safety: If any exception is thrown by the destruction of the old value of `this`, `this->empty()` will be `true` and no move assignment will take place. If `rhs.empty()` is false and an exception is thrown during the call to `T_j`'s move constructor (with `j` being `rhs.index()`), `this->empty()` will be `true` and no copy assignment will take place. If `rhs.empty()` is false and an exception is thrown during the call to `T_i`'s move assignment, the state of the contained value and `t`'s contained value are as defined by the exception safety guarantee of `T_i`'s copy assignment; `this->index()` will be `j`.

`template <class T, class... Args> void emplace(Args&&...)`

Requires: `is_constructible<T, Args&&...>::value` is `true`.
Effects: Destructs the currently contained value. Then initializes the contained value as if constructing a value of type `T` with the arguments `std::forward<Args>(args)...`.
Postcondition: `holds_alternative<T>(*this)` is `true`.
Throws: Any exception thrown by the selected constructor of `T`.
Exception safety: If any exception is thrown by the destruction of the old value of `this`, `this->empty()` will be `true` and no construction of `T` will

take place. If an exception is thrown during the call to `T`'s constructor, `this->empty()` will be `true`.

`template <class T, class U, class... Args> void emplace(initializer_list<V> li, Args&&...)`

Requires: `is_constructible<T, initializer_list<U>&, Args&&...>::value` is `true`.

Effects: Destructs the currently contained value. Then initializes the contained value as if constructing an object of type `T` with the arguments `il, std::forward<Args>(args)...`.

Postcondition: `holds_alternative<T>(*this)` is `true`

Throws: Any exception thrown by the selected constructor of `T`.

Exception safety: If any exception is thrown by the destruction of the old value of `this`, `this->empty()` will be `true` and no construction of `T` will take place. If an exception is thrown during the call to `T`'s constructor, `this->empty()` will be `true`.

Remarks: The function shall not participate in overload resolution unless `is_constructible<T, initializer_list<U>&, Args&&...>::value` is `true`.

`void clear()`

Effects: destructs the contained value (if not empty) and marks the `variant` as empty.

Postcondition: `empty()` is `true`.

Throws: Any exception thrown by any `~T_i()` for all `i`.

`bool empty() const`

Effects: returns whether the `variant` contains a value.

`size_t index() const`

Requires: `empty()` is `false`

Effects: Returns the index `j` of the first match of the contained value's type `T_j` in the `variant`'s template parameter list.

`void swap(variant& rhs) noexcept; //see below`

Requires: LValues in `T_i` shall be swappable and `is_move_constructible<T_i>::value` is `true` for all `i`.

Effects: calls `swap(get<T_j>(), rhs->get<T_j>(rhs))` with `j` being `this->index()`.

Throws: Any exceptions that the expression in the Effects clause throws, including `bad_variant_access`.

Exception safety: If an exception is thrown during the call to function `swap` the state of the value of `this` and of `rhs` is determined by the exception safety guarantee of `swap` for lvalues of `T_j` with `j` being `this->index()`. If an exception is thrown during the call to `T_j`'s move constructor, the state of the value of `this` and of `rhs` is determined by the exception safety guarantee of `T_j`'s move constructor.

In-place construction

```
struct in_place_t{};
constexpr in_place_t in_place{};
```

The struct `in_place_t` is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, `variant<Types...>` has a constructor with `in_place_t` as the first argument followed by an argument pack; this indicates that `T` should be constructed in-place (as if by a call to placement new expression) with the forwarded argument pack as parameters.

`class bad_variant_access`

```
class bad_variant_access : public logic_error {
public:
  explicit bad_variant_access(const string& what_arg);
  explicit bad_variant_access(const char* what_arg);
};
```

The class `bad_variant_access` defines the type of objects thrown as exceptions to report the situation where an attempt is made to access the value of a `variant` object `v` through one of the `get` overloads in an invalid way: * for `get` overloads with template parameter list `size_t I, class... Types`, because `I` is out of range, * for `get` overloads with template parameter list `class T, class... Types`, because `is_alternative<T>(v)` is `false`

`bad_variant_access(const string& what_arg)`

Effects: Constructs an object of class `bad_variant_access`.

`bad_variant_access(const char* what_arg)`

Effects: Constructs an object of class `bad_variant_access`.

`variant` helper classes

`variant_num_alternatives`

Identical to `tuple_size`, only for `variant` instead of `tuple`.

Should this be unified?

`variant_alternative`

Identical to `tuple_element`, only for `variant` instead of `tuple`.

Should this be unified?

Handling of alternatives

```
template <class T, class... Types> constexpr bool is_alternative(const
variant<Types...>& v) noexcept
```

Effects: returns `true` if `v` can hold a value of `T`; `false` otherwise.

```
template <class T, class... Types> constexpr size_t alternative_index(const
variant<Types...>& v) noexcept;
```

Effects: returns the first `i` for which `is_alternative<T_i>(v)` is `true`. If no such `i` exists, returns `alternative_unavailable`.

Value access

```
template <class T, class... Types> bool holds_alternative(const
variant<Types...>& v) noexcept;
```

Effects: returns `true` if `v.empty()` is `false` and `v` currently holds a value of type `T`, i.e. if it `get<T>(v)` will not throw. It thus only returns `true` if `is_alternative<T>(v)` is `true`.

```
template <size_t I, class... Types> variant_alternative_t<I,
variant<Types...>>& get(variant<Types...>& v);
```

Requires: `I < sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.
Effects: returns a reference to the contained value of type `T_i` with `i` being `I`.
Throws: `bad_variant_access` if `v.empty()` or `v.index() != I`.

```
template <size_t I, class... Types> variant_alternative_t<I,
variant<Types...>>&& get(variant<Types...>&& v);
```

Requires: `I < sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.
Effects: Equivalent to `return std::forward<typename valiant_alternative<I, variant<Types...> >::type&&>(get<I>(v));`.
Throws: `bad_variant_access` if `v.empty()` or `v.index() != I` plus any exception that the expression in the Effects clause throws.

[Note: if a `T` in `Types` is some reference type `X&`, the return type is `X&`, not `X&&`. However, if the element type is a non-reference type `T`, the return type is `T&&`.]

```
template <size_t I, class... Types> const variant_alternative_t<I,
variant<Types...>>& get(const variant<Types...>&) v;
```

Requires: `I < sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.
Effects: A const reference to the contained value of type `T_i` with `i` being `I`.
Throws: `bad_variant_access` if `v.empty()` or `v.index() != I`.

```
template <class T, class... Types> T& get(variant<Types...>& v)
```

Requires: `is_alternative<T>(v)`. The program is ill-formed if not.
Effects: Equivalent to `return get<alternative_index<T>(v)>(v)`.
Throws: Any exceptions that the expression in the Effects clause throws.

```
template <class T, class... Types> T&& get(variant<Types...>&&
v)
```

Requires: `is_alternative<T>(v)`. The program is ill-formed if not.
Effects: Equivalent to `return get<alternative_index<T>(v)>(v)`.
Throws: Any exceptions that the expression in the Effects clause throws.

`template <class T, class... Types> const T& get(const variant<Types...>&)`

Requires: `is_alternative<T>(v)`. The program is ill-formed if not.
Effects: Equivalent to `return get<alternative_index<T>(v)>(v)`.
Throws: Any exceptions that the expression in the Effects clause throws.

Relational operators

`template <class... TTypes, class... UTypes> bool operator==(const variant<TTypes...>& v, const variant<UTypes...>& w)`

Requires: For all combinations of `i` and `j`, where `0 <= i` and `i < sizeof...(TTypes)`, and `0 <= j` and `j < sizeof...(UTypes)`, `get<i>(v) == get<j>(w)` is a valid

expression returning a type that is convertible to `bool`. Returns: : `true` if `v.empty() && w.empty()`; `false` if `v.empty() != w.empty()`. In all other cases, returns `true` if `get<i>(v) == get<j>(w)` with `i` being `v.index()` and `j` being `w.index()`, otherwise `false`.

`template <class... TTypes, class... UTypes> bool operator!=(const variant<TTypes...>& v, const variant<UTypes...>& w)`

Returns: `!(v == w)`.

`template <class... TTypes, class... UTypes> bool operator<(const variant<Types...>& v, const variant<Types...>& w)`

Requires: For all combinations of `i` and `j`, where `0 <= i` and `i < sizeof...(TTypes)`, and `0 <= j` and `j < sizeof...(UTypes)`, `get<i>(v) < get<j>(w)` is a valid

expression returning a type that is convertible to `bool`.

Returns: `false` if `v.empty() && w.empty()`; `false` if `v.empty() != w.empty()`. In all other cases, returns `true` if `get<i>(v) < get<j>(w)` with `i` being `v.index()` and `j` being `w.index()`, otherwise `false`.

```
template <class... TTypes, class... UTypes> bool operator>(const
variant<Types...>& v, const variant<Types...>& w)
```

Requires: For all combinations of `i` and `j`, where `0 <= i` and `i < sizeof...(TTypes)`, and `0 <= j` and `j < sizeof...(UTypes)`, `get<i>(v) > get<j>(w)` is a valid

expression returning a type that is convertible to `bool`.

Returns: `false` if `v.empty() && w.empty()`; `false` if `v.empty() != w.empty()`. In all other cases, returns `true` if `get<i>(v) > get<j>(w)` with `i` being `v.index()` and `j` being `w.index()`, otherwise `false`.

```
template <class... TTypes, class... UTypes> bool operator<=(const
variant<Types...>& v, const variant<Types...>& w)
```

Returns: `(v < w || v == w)`.

```
template <class... TTypes, class... UTypes> bool operator<=(const
variant<Types...>& v, const variant<Types...>& w)
```

Returns: `(v > w || v == w)`.

Specialized algorithms

```
template <class... Types> void swap(variant<Types...>& x, variant<Types...>&
y) noexcept(see below)
```

Effects: calls `x.swap(y)`.

Hash support

```
template <class... Types> struct hash<experimental::variant<Types...>>
```

Requires: the template specialization `hash<T_i>` shall meet the requirements of class template `hash` (C++11 §20.8.12) for all `i`.

The template specialization `hash<variant<Types...>>` shall meet the requirements of class template `hash`. For an object `o` of type `variant<Types...>`, if `o.empty() == false` and with `o.index()` being `j`, `hash<variant<Types...>>()(o)` shall evaluate to the same value as `hash<T_j>()(get<T_j>(o))`.

Conclusion

A variant has proven to be a useful tool. This paper proposes the necessary, basic ingredients.

Acknowledgments

We would like to thank Vincenzo Innocente and Philippe Canal for their comments on this proposal.

References

1. *Working draft, technical specification on c++ extensions for library fundamentals*. N3848

2. *Boost.variant* [online]. Available from: http://www.boost.org/doc/libs/1_56_0/doc/html/variant.html