Variant: a type-safe union without undefined behavior (v2).

P0087R0, ISO/IEC JTC1 SC22 WG21

Axel Naumann (axel@cern.ch), 2015-09-28

Contents

Introduction 3

Version control 4

Discussion 4
   Empty state and default construction ......................... 4
   Feature Test ..................................................... 4

Variant Objects 4
   In general ......................................................... 4
   Changes to header <tuple> ..................................... 5
   Header <variant> synopsis .................................... 5
   Class template variant .......................................... 7
      Construction .................................................. 9
      Destructor .................................................... 12
      Assignment ................................................... 12
      Modifiers ..................................................... 14
   bool valid() const noexcept .................................. 16
   size_t index() const noexcept ................................ 16
   void swap(variant& rhs) noexcept(see below) .............. 16
In-place construction .............................................. 16

class bad_variant_access ........................................ 17
  bad_variant_access(const string& what_arg) .............. 17
  bad_variant_access(const char* what_arg) .............. 17

tuple interface to class template variant ...................... 18

  template <class T, class... Types> struct tuple_size
              <variant<Types...>> ................................ 18

  template <size_t I, class... Types> struct tuple_element<I,
              variant<Types...>> ................................ 18

Value access ........................................................ 18

  template <class T, class... Types> bool holds_alternative(const
              variant<Types...>& v) noexcept; .................... 18

  template <class T, class... Types> remove_reference_t<T>&
              get(variant<Types...>& v) ..................... 18

  template <class T, class... Types> const remove_reference_t<T>&
              get(const variant<Types...>&) .................. 18

  template <class T, class... Types> T&& get(variant<Types...>&&
              v) ..................................................... 18

  template <size_t I, class... Types> remove_reference_t<T>&
              get(variant<Types...>& v) ..................... 19

  template <size_t I, class... Types> const remove_reference_t<T>&
              get(const variant<Types...>& v) .................. 19

  template <size_t I, class... Types> T&& get(variant<Types...>&&
              v) ..................................................... 19

  template <class T, class... Types> remove_reference_t<T>*
              get(variant<Types...>* v) ...................... 19

  template <class T, class... Types> const remove_reference_t<T>*
              get(const variant<Types...>* v) .................. 19

  template <size_t I, class... Types> remove_reference_t<
              tuple_element_t<I, variant<Types...>>>
              get(variant<Types...>* v) ...................... 19

  template <size_t I, class... Types> const remove_reference_t<
              tuple_element_t<I, variant<Types...>>>
              get(const variant<Types...>* v) .......... 19

Relational operators ............................................. 20

  template <class... Types> bool operator==(const
              variant<Types...>& v, const variant<Types...>& w) 20
C++ needs a type-safe union; here is a proposal. It attempts to apply the lessons learned from `optional` (1). It behaves as below:

```cpp
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&) {}
types given to `variant`. These template arguments are called alternatives.

**Changes to header `<tuple>`**

`variant` employs the meta-programming facilities provided by the header `tuple`. It requires one additional facility:

```cpp
static constexpr const size_t tuple_not_found = (size_t) -1;
template <class T, class U> class tuple_find; // undefined
template <class T, class U> class tuple_find<T, const U>;
template <class T, class U> class tuple_find<T, volatile U>;
template <class T, class U> class tuple_find<T, const volatile U>;
template <class T, class... Types> class tuple_find<T, tuple<Types...>>;
template <class T, class T1, class T2> class tuple_find<T, pair<T1, T2>>;
template <class T, class... Types> class tuple_find<T, variant<Types...>>;
```

The `cv`-qualified versions behave as re-implementations of the non-`cv`-qualified version. The last versions are defined as

```cpp
template <class T, class... Types>
class tuple_find<T, tuple<Types...>>:
  integral_constant<std::size_t, INDEX> {};

template <class T, class T1, class T2>
class tuple_find<T, pair<T1, T2>>:
  public tuple_find<T, tuple<T1, T2>> {};

template <class T, class... Types>
class tuple_find<T, variant<Types...>>:
  public tuple_find<T, tuple<Types...>> {};
```

where `INDEX` is the index of the first occurrence of `T` in `Types...` or `tuple_not_found` if the type does not occur. `tuple_find` is thus the inverse operation of `tuple_index`: for any tuple type `T` made up of different types, `tuple_index_t<tuple_find<U, T>::value>` is `U` for all of `T`'s parameter types.

**Header `<variant>` synopsis**

```cpp
namespace std {
  namespace experimental {
    inline namespace fundamentals_vXXXX {
      // 2.?, variant of value types
      template <class... Types> class variant;
    }
  }
}
```
// 2.?, In-place construction
template <class T> struct emplaced_type_t{};
template <class T> constexpr emplaced_type_t<T> emplaced_type;

template <size_t I> struct emplaced_index_t{};
template <size_t I> constexpr emplaced_index_t<I> emplaced_index;

// 2.?, class bad_variant_access
class bad_variant_access;

// 2.?, tuple interface to class template variant
template <class T> class tuple_size;
template <size_t I, class T> class tuple_element;
template <class T, class... Types>
    struct tuple_size<variant<Types...>>;
template <size_t I, class... Types>
    struct tuple_element<I, variant<Types...>>;

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

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

template <size_t I, class... Types>
    remove_reference_t<tuple_element_t<I, variant<Types...>>>&
        get(variant<Types...>&);
template <size_t I, class... Types>
    tuple_element_t<I, variant<Types...>>&
        get(variant<Types...>&&);
template <size_t I, class... Types>
    remove_reference_t<const tuple_element_t<I, variant<Types...>>>&
        get(const variant<Types...>&);

template <class T, class... Types>
    remove_reference_t<T>* get_if(variant<Types...>*);
template <class T, class... Types>
    const remove_reference_t<T>* get_if(const variant<Types...>*);
remove_reference_t<tuple_element_t<I, variant<Types...>>>*  
get_if(variant<Types...>*);

template <size_t I, class... Types>
const remove_reference_t<tuple_element_t<I, variant<Types...>>>*
get_if(const variant<Types...>*);

// 2.?, relational operators

template <class... Types>
bool operator==(const variant<Types...>&, const variant<Types...>&);

template <class... Types>
bool operator!=(const variant<Types...>&, const variant<Types...>&);

template <class... Types>
bool operator<(const variant<Types...>&, const variant<Types...>&);

template <class... Types>
bool operator>(const variant<Types...>&, const variant<Types...>&);

template <class... Types>
bool operator<=(const variant<Types...>&, const variant<Types...>&);

template <class... Types>
bool operator>=(const variant<Types...>&, const variant<Types...>&);

// 2.?, Visitation

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

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

} // namespace fundamentals_vXXXX
} // namespace experimental

// 2.?, Hash support

template <class T> 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 {
    public:

        // 2.3 variant construction
        constexpr variant() noexcept;
        variant(const variant&) noexcept(see below);
        variant(variant&&) noexcept(see below);

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

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

        // 2.3. 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&&);

        // 2.3. `variant` assignment
        variant& operator=(const variant&);
        variant& operator=(variant&&) noexcept(see below);

        template <class T> variant& operator=(const T&);
        template <class T> variant& operator=(const T&&) noexcept(see below);
    } // variant
}; // namespace fundamentals_vXXXX
// 2.?, `variant` modifiers
void clear() noexcept;

template <class T, class... Args> void emplace(Args&&...);
template <class T, class U, class... Args>
    void emplace(initializer_list<U>, Args&&...);
template <size_t I, class... Args>
    void emplace(Args&&...);

d // 2.?, value status
bool valid() const noexcept;
size_t index() const noexcept;

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

private:
    static constexpr size_t max_alternative_sizeof
        = ...; // exposition only
    char storage[max_alternative_sizeof]; // exposition only
    size_t value_type_index; // exposition only
}; // namespace fundamentals_vXXXX
} // namespace experimental
} // namespace std

Any instance of variant<Types...> at any given time either contains a value of
one of its template parameter Types, or is in an invalid state. When an instance
of variant<Types...> contains a value of alternative type T, it means that an
object of type T, referred to as the variant<Types...> object’s contained value,
is allocated within the storage of the variant<Types...> object. Implementa-
tions are not permitted to use additional storage, such as dynamic memory, to
allocate its contained value. The contained value shall be allocated in a region
of the variant<Types...> storage suitably aligned for all types in Types.

All types in Types shall be object types and shall satisfy the requirements of
Destructible (Table 24).

Construction

For the default constructor, an exception is thrown if the first alternative type
throws an exception. For all other variant constructors, an exception is thrown
only if the construction of one of the types in Types throws an exception.
The copy and move 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 move and copy constructor of `variant<>` shall be `constexpr` functions.

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

### `constexpr variant()` noexcept

**Effects:** Constructs a `variant` in invalid state.

**Postconditions:** `valid()` is `false`.

**Remarks:** The expression inside `noexcept` is equivalent to `is_nothrow_default_constructible_v<T_0>`. The function shall not participate in overload resolution if `is_default_constructible_v<T_0>` is `false`.

### `variant(const variant& w)`

**Requires:** `is_copy_constructible_v<T_i>` is `true` for all \( i \).

**Effects:** initializes the `variant` to hold the same alternative as \( w \). 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_v<T_i>` 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 \( w.index() \).

**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_v<T_i>::value` for all \( i \).

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

**Requires:** `is_copy_constructible_v<T>` 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:** The function shall not participate in overload resolution unless `T` is one of `Types`... The function shall be `= delete` if there are multiple occurrences of `T` in `Types`... If `T`'s selected constructor is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.
template <class T> constexpr variant(T&& t)

Requires: is_move_constructible_v<T> 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: The function shall not participate in overload resolution unless T is one of Types... The function shall be = delete if there are multiple occurrences of T in Types... If T’s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

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

Requires: T is one of Types.... is_constructible_v<T, Args&&...> 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: The function shall be = delete if there are multiple occurrences of T in Types.... 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(emplaced_type_t<T>, initializer_list<U> il, Args&&...);

Requires: T is one of Types.... is_constructible_v<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 be = delete if there are multiple occurrences of T in Types.... If T’s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

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

Requires: I must be less than sizeof...(Types). is_constructible_v<tuple_element_t<I, variant>, Args&&...> is true.
Effects: Initializes the contained value as if constructing an object of type tuple_element_t<I, variant> with the arguments std::forward<Args>(args)....
Postcondition: index() is I
Throws: Any exception thrown by the selected constructor of tuple_element_t<\( I \), variant>.
Remarks: If tuple_element_t<\( I \), variant>’s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

\[
\text{template } \langle \text{size_t } I, \text{ class } U, \text{ class... Args} \rangle \text{ constexpr explicit}
\]
\[
\text{variant(emplaced_index_t}<I>, \text{ initializer_list}<U> \text{ il, Args&&...});
\]

Requires: \( I \) must be less than sizeof...(Types). is_constructible_v<tuple_element_t<\( I \), variant>, initializer_list<U>&, Args&&...> is true.
Effects: Initializes the contained value as if constructing an object of type tuple_element_t<\( I \), variant> with the arguments il, std::forward<Args>(args)....
Postcondition: index() is I
Remarks: The function shall not participate in overload resolution unless is_constructible_v<tuple_element_t<\( I \), variant>, initializer_list<U>&, Args&&...> is true. If tuple_element_t<\( I \), variant>’s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

Destructor

-variant()

Effects: Behaves as if clear() is invoked.

Assignment

variant& operator=(const variant& rhs)

Requires: is_copy_constructible_v<T_i> && is_copy_assignable_v<T_i> is true for all \( i \).
Effects: If index() == rhs.index(), calls get\( j \)(*this) = get\( j \)(rhs) with \( j \) being index(). Else copies the value contained in rhs to a temporary, then destructs the current contained value of *this. Sets *this to contain the same type as rhs and move-constructs the contained value from the temporary.
Returns: *this.
Postconditions: index() == rhs.index()
Exception safety: If an exception is thrown during the call to T_i’s copy constructor (with \( i \) being rhs.index()), *this will remain unchanged. If an exception is thrown during the call to T_i’s move constructor, valid()
will be \texttt{false} and no copy assignment will take place; the \texttt{variant} will be in a valid but partially unspecified state. If an exception is thrown during the call to \texttt{T\_i}'s copy assignment, the state of the contained value is as defined by the exception safety guarantee of \texttt{T\_i}'s copy assignment; \texttt{index()} will be \texttt{i}.

\textbf{variant\& operator=(const variant\&\& rhs) noexcept}(see below)

\textbf{Requires:} \texttt{is\_move\_constructible\_v<T\_i> \&\& is\_move\_assignable\_v<T\_i>}
\textbf{is} \texttt{true} for all \texttt{i}.

\textbf{Effects:} \texttt{If valid()} \texttt{\&\& index() == rhs.index()}, the move-assignment operator is called to set the contained object to \texttt{std::forward<T\_j>(get<j>(rhs))} with \texttt{j} being \texttt{rhs.index()}. Else destructs the current contained value of \texttt{*this} if \texttt{valid()} \texttt{is true}, then initializes \texttt{*this} to hold the same alternative as \texttt{rhs} and initializes the contained value with \texttt{std::forward<T\_j>(get<j>(rhs))}.

\textbf{Returns:} \texttt{*this}.

\textbf{Remarks:} The expression inside \texttt{noexcept} is equivalent to: \texttt{is\_nothrow\_move\_assignable\_v<T\_i> \&\& is\_nothrow\_move\_constructible\_v<T\_i> for all i}.

\textbf{Exception safety:} If an exception is thrown during the call to \texttt{T\_j}'s move constructor (with \texttt{j} being \texttt{rhs.index()}), \texttt{valid()} will be \texttt{false} and no move assignment will take place; the \texttt{variant} will be in a valid but partially unspecified state. If an exception is thrown during the call to \texttt{T\_j}'s move assignment, the state of the contained value is as defined by the exception safety guarantee of \texttt{T\_j}'s move assignment; \texttt{index()} will be \texttt{j}.

\texttt{template <class T> variant\& operator=(const T\& t)}

\texttt{template <class T> variant\& operator=(const T\&\& t) noexcept}(see below)

\textbf{Requires:} The overload set \texttt{T\_i(t)} of all constructors of all alternatives of this \texttt{variant} must resolve to exactly one best matching constructor call of an alternative type \texttt{T\_j}, according to regular overload resolution; otherwise the program is ill-formed. [Note:

\begin{verbatim}
variant<string, string> v;
v = "abc";
\end{verbatim}

is ill-formed, as both alternative types have an equally viable constructor for the argument.]
Effects: If *this holds a T_j, the copy / move assignment operator is called, passing t. Else, for the copy assignment and if is_move_constructible<T_j> is true, creates a temporary of type T_j, passing t as argument to the selected constructor. Destroys the current contained value of *this, initializes *this to hold the alternative T_j, and initializes the contained value, for the move assignment by calling the selected constructor overload, passing t; for the copy-assignment by move-constructing the contained value from the temporary if is_move_constructible<T_j> is true, and copy-constructing the contained value passing t if is_move_constructible<T_j> is false.

Postcondition: holds_alternative<T_j>(*this) is true.

Returns: *this.

Exception safety: If an exception is thrown during the call to the selected constructor, valid() will be false and no copy / move assignment will take place. If an exception is thrown during the call to T_j’s copy / move assignment, the state of the contained value and t are as defined by the exception safety guarantee of T_j’s copy / move assignment; valid() will be true.

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.

Modifiers

void clear() noexcept

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

Postcondition: valid() is false.

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

Requires: is_constructible_v<T, Args&&...> is true.

Effects: Destroys the currently contained value if valid() is true. 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 an exception is thrown during the call to T’s constructor, valid() will be false; the variant will be in a valid but partially unspecified state.

template <class T, class U, class... Args> void emplace(initializer_list<U> il, Args&&...)

14
**Variant Objects**

**Requires:** `is_constructible_v<T, initializer_list<U>&, Args&&...>` is true.

**Effects:** Destructs the currently contained value if `valid()` is true. 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 an exception is thrown during the call to `T`’s constructor, `valid()` will be `false`; the `variant` will be in a valid but partially unspecified state.

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

```cpp
template <size_t I, class... Args> void emplace(Args&&...)
```

**Requires:** `is_constructible_v<tuple_element<I, variant>, Args&&...>` is true.

**Effects:** Destructs the currently contained value if `valid()` is true. Then initializes the contained value as if constructing a value of type `tuple_element<I, variant>` with the arguments `std::forward<Args>(args)...`

**Postcondition:** `index()` is `I`.

**Throws:** Any exception thrown by the selected constructor of `tuple_element<I, variant>`.

**Exception safety:** If an exception is thrown during the call to `tuple_element<I, variant>`’s constructor, `valid()` will be `false`; the `variant` will be in a valid but partially unspecified state.

```cpp
template <size_t I, class U, class... Args> void emplace(initializer_list<U> il, Args&&...)
```

**Requires:** `is_constructible_v<tuple_element<I, variant>, initializer_list<U>&, Args&&...>` is true.

**Effects:** Destructs the currently contained value if `valid()` is true. Then initializes the contained value as if constructing an object of type `tuple_element<I, variant>` with the arguments `il, std::forward<Args>(args)...`

**Postcondition:** `index()` is `I`.

**Throws:** Any exception thrown by the selected constructor of `tuple_element<I, variant>`.

**Exception safety:** If an exception is thrown during the call to `tuple_element<I, variant>`’s constructor, `valid()` will be `false`; the `variant` will be in a valid but partially unspecified state.
Remarks: The function shall not participate in overload resolution unless
\( \text{is}\_\text{constructible}\_\text{v}<\text{tuple}\_\text{element}<I,\ \text{variant}>, \ \text{initializer}\_\text{list}<U>&, \ \text{Args}&&... > \text{is true.} \)

bool valid() const noexcept

Effects: returns whether the variant contains a value (returns true), or is in
a valid but partially unspecified state (returns false).

size_t index() const noexcept

Effects: Returns the index \( j \) of the currently active alternative, or
\text{tuple}\_\text{not}\_\text{found} if valid() is false.

void swap(variant& rhs) noexcept(see below)

Requires: valid() && rhs.valid(). \text{is}\_\text{move}\_\text{constructible}\_\text{v}<T_i> is
true for all \( i \).

Effects: if index() == rhs.index(), calls swap(get<i>(*this), get<i>(hrs))
with \( i \) being index(). Else calls swap(*this, hrs).

Throws: Any exceptions that the expression in the Effects clause throws.

Exception safety: If an exception is thrown during the call to function
\text{swap}(\text{get}<i>(\ast this), \text{get}<i>(hrs)), the state of the value of \text{this}
and of rhs is determined by the exception safety guarantee of \text{swap}
for lvalues of \( T_i \) with \( i \) being index(). If an exception is thrown during
the call to \text{swap}(*this, hrs), the state of the value of \text{this} and of
rhs is determined by the exception safety guarantee of \text{variant}'s move
constructor and assignment operator.

In-place construction

\text{template <class T> struct emplaced}\_\text{type}\_\text{t}{};
\text{template <class T> constexpr emplaced}\_\text{type}\_\text{t}<T> emplaced\_\text{type}{};
\text{template <size_t I> struct emplaced}\_\text{index}\_\text{t}{};
\text{template <size_t I> constexpr emplaced}\_\text{index}\_\text{t}<I> emplaced\_\text{index}{};

Template instances of \text{emplaced}\_\text{type}\_\text{t} are empty structure types used as
unique types to disambiguate constructor and function overloading, and sig-
ingaling (through the template parameter) the alternative to be constructed.
Specifically, \text{variant}<\text{Types}...> has a constructor with \text{emplaced}\_\text{type}\_\text{t}<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 a placement new expression) with
the forwarded argument pack as parameters. If a variant’s types has multiple
occurrences of $T$, `emplaces_index_t` must be used.

Template instances of `emplaced_index_t` are empty structure types used as
unique types to disambiguate constructor and function overloading, and signaling
(through the template parameter) the alternative to be constructed. Specifically,
`variant<Types...>` has a constructor with `emplaced_index_t<I>` as the first
argument followed by an argument pack; this indicates that `tuple_element<I, variant>`
should be constructed in-place (as if by a call to a placement new expression) with the forwarded argument pack as parameters.

```cpp
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 through one of the `get` or `visit` overloads in an invalid way:

- for `get` overloads with template parameter list `size_t I, class... Types`, because $I$ does not equal to `index()
- for `get` overloads with template parameter list `class T, class... Types`, because `holds_alternative<T>(v)` is `false`
- for `visit` overloads with any variant argument for which `valid()` is `false`

The value of `what_arg` of an exception thrown in these cases is implementation
defined.

```cpp
bad_variant_access(const string& what_arg)
```

**Effects**: Constructs an object of class `bad_variant_access`.

```cpp
bad_variant_access(const char* what_arg)
```

**Effects**: Constructs an object of class `bad_variant_access`.


tuple interface to class template variant

```cpp
template <class T, class... Types> struct tuple_size<variant<Types...>> { }
```

```cpp
template <class... Types>
class tuple_size<variant<Types...>>
  : public integral_constant<size_t, sizeof...(Types)> { }
```

```cpp
template <size_t I, class... Types>
struct tuple_element<I, variant<Types...>>
  : public tuple_element<I, tuple<Types...>> { }
```

Value access

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

**Requires:** The type T occurs exactly once in Types... Otherwise, the program is ill-formed.

**Effects:** returns true if index() is equal to tuple_find<T, variant<Types...>>.

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

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

**Requires:** The type T occurs exactly once in Types... Otherwise, the program is ill-formed.

**Effects:** Equivalent to return get<tuple_find<T, variant<Types...>>::value>(v).

**Throws:** Any exceptions that the expression in the Effects clause throws.

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

**Requires:** The type T occurs exactly once in Types... Otherwise, the program is ill-formed.

**Effects:** Equivalent to return get<tuple_find<T, variant<Types...>>::value>(v).

**Throws:** Any exceptions that the expression in the Effects clause throws.

**Remarks:** if the element type T 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> remove_reference_t<T>& get(variant<Types...>& v)

template <size_t I, class... Types> const remove_reference_t<T>& get(const variant<Types...>& v)

Requires: The program is ill-formed unless I < sizeof...(Types).
Effects: Return a (const) reference to the object stored in the variant, if v.index() is I, else throws an exception of type bad_variant_access.
Throws: An exception of type bad_variant_access.

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

Requires: The program is ill-formed unless I < sizeof...(Types).
Effects: Equivalent to return std::forward<typename tuple_element<I, variant<Types...> >::type&&>(get<I>(v)).
Throws: Any exceptions that the expression in the Effects clause throws.
Remarks: if the element type typename tuple_element<I, variant<Types...> >::type 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 <class T, class... Types> remove_reference_t<T>* get(variant<Types...>* v)

template <class T, class... Types> const remove_reference_t<T>* get(const variant<Types...>* v)

Requires: The type T occurs exactly once in Types.... Otherwise, the program is ill-formed.
Effects: Equivalent to return get<tuple_find<T, variant<Types...>>::value>(v).

template <size_t I, class... Types> remove_reference_t<tuple_element_t<I, variant<Types...>>>* get(variant<Types...>*)

template <size_t I, class... Types> const remove_reference_t<tuple_element_t<I, variant<Types...>>>* get(const variant<Types...>*)

Requires: The program is ill-formed unless I < sizeof...(Types).
Effects: Return a (const) reference to the object stored in the variant, if v->index() is I, else returns nullptr.
Relational operators

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

Requires: get<i>(v) == get<i>(w) is a valid expression returning a type that is convertible to bool, for all i in 0 ... sizeof...(Types).
Returns: true if !v.valid() && !w.valid(). Otherwise, true if v.index() == w.index() && get<i>(v) == get<i>(w) with i being v.index(), otherwise false.

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

Returns: !(v == w).

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

Requires: get<i>(v) < get<i>(w) is a valid expression returning a type that is convertible to bool, for all i in 0 ... sizeof...(Types).
Returns: false if !v.valid() && !w.valid(). Otherwise, true if v.index() < w.index() || (v.index() == w.index() && get<i>(v) < get<i>(w)) with i being v.index(), otherwise false.

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

Returns: w < v.

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

Returns: !(v > w).

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

Returns: !(v < w).
Visitation

template <class Visitor, class... Variants> decltype(auto) visit(Visitor& vis, Variants&... vars)

template <class Visitor, class... Variants> decltype(auto) visit(const Visitor& vis, const Variants&... vars)

Requires: The expression in the Effects clause must be a valid expression of the same type, for all combinations of alternative types of all variants.
Effects: Calls vis(get<T_0_i>(get<0>(vars)),get<T_1_i>(get<1>(vars),...)
         with T_j_i being get<j>(vars).index().
Throws: If var.valid() is false for any var in vars, throws an exception of type bad_variant_access.
Remarks: Visit with sizeof...(Variants) being 0 is ill-formed. For sizeof...(Variants) being 1, the invocation of the callable must be implemented in O(1), i.e. it must must not depend on sizeof...(Types).
For sizeof...(Variants) greater 1, the invocation of the callable has no complexity requirements.

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.

Conclusion

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

Acknowledgments

Thank you, Nevin “:-)” Liber, for bringing sanity to this proposal. Agustín K-balío Bergé and Antony Polukhin provided very valuable feedback, criticism and suggestions. Thanks also to Vincenzo Innocente and Philippe Canal for their comments.
References