Variant: a type-safe union (v3).

N4516, ISO/IEC JTC1 SC22 WG21

Axel Naumann (axel@cern.ch), 2015-05-06

Contents

Variant is the very spice of life,
That gives it all its flavor.
- William Cowper's "The Task", or actually a variant thereof

Introduction

C++ needs a type-safe 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&) {}
```

Version control

Results of the LEWG review in Urbana

The LEWG review in Urbana resulted in the following straw polls that motivated changes in this revision of the paper:

- Should we use a `tuple`-like interface instead of the collection of `variant`-specific functions, `is_alternative` etc.? SF=8 WF=5 N=2 WA=1 SA=0
- Consent: `variant` should be as `constexpr` as `std::optional`
- Consent: The paper should discuss the never-empty guarantee
- Consent: Expand on `variant<int, int>` and `variant<int, const int>`.
- Visitors are needed for the initial variant in the TS? SF=4 WF=3 N=5 WA=4 SA=0
- Recursive variants are needed? SF=0 WF=0 N=8 WA=4 SA=2

Results of the LEWG review in Lenexa

In Lenexa, LEWG decided that `variant` should model a discriminated union.

- Approval votes on emptiness:
- empty, queryable state: 12
- invalid, assignable, UB on read: 13
- invalid, throws on read: 6
- double buffer: 5

- require all members nothrow-move-constructible: 1
- require either move-noexcept or one-default-construct-noexcept: 0
- Want to query whether in empty state: SF=4 WF=4 N=4 WA=1 SA=1
- Should the default constructor lead to the empty state? SF=3 WF=1 N=3 WA=1 SA=5; later SF=2 WF=0 N=2 WA=1 SA=6
- Should the default constructor try to construct the first element? SF=5 WF=3 N=1 WA=2 SA=2, later SF=6 WF=3 N=0 WA=1 SA=1
- Should the default constructor search for a default-constructible type and take the first possible one? (no earlier poll), later SF=0 WF=1 N=2 WA=5 SA=3
- Remove heterogeneous assignment? SF=9 WF=5 N=3 WA=0 SA=1
- Remove conversions, e.g. `variant<int, string> x = "abc";`? SF=5 WF=4 N=1 WA=1 SA=0
- Allow `variant<string> == const char *` and `variant<const char *, string> == const char *`? SF=0 WF=2 N=5 WA=3 SA=3
- Allow `variant<string> == variant<const char *>`, and `variant<A, B, C> == variant<X, Y, Z>`? SF=0 WF=1 N=0 WA=4 SA=8
- Allow `variant<int, const int>`, qualified types in general? SF=9 WF=4 N=1 WA=1 SA=1
- Allow types to be reference types? SF=6 WF=4 N=6 WA=1 SA=0
- Allow void? SF=6 WF=9 N=2 WA=0 SA=0
- Provide multi-visitation `visit(VISITOR, var1, var2, var3, ...)`? SF=0 WF=7 N=7 WA=1 SA=0
- Provide binary visitation `visit(VISITOR, v1, v2)`? SF=0 WF=1 N=10 WA=1 SA=3
- Approval vote of visitor return types:
- `common_type`: 12
- require same return type: 13
- return type of `op()()`, rest must convert to that: 1
- `variant<return types>`: 2
- `variant<return types>` if they're different, otherwise single return type: 0

- no `void * data()`
- yes `T* get<T>(variant<A, B, C> *)` (a la `any_cast`)
- Should `index()` return -1 on empty? (The alternative is to make non-emptiness a precondition.) SF=4 WF=1 N=3 WA=1 SA=2
- Should `variant::{visit,get}` have preconditions that the `variant` not be empty? SF=4 WF=8 N=2 WA=0 SA=0

Differences to revision 1 (N4218)

As requested by the LEWG review in Urbana, this revision

- considerably expands the discussion of why this proposal allows the `variant` to be empty;
- explains how duplicate (possibly *cv*-qualified) types and `void` as alternatives behave;
- reuses (and extends, for consistency) the facilities provided by `tuple` for parameter pack operations; `is_alternative` does not yet exist as part of `tuple` and is thus kept;
- employs the "perfect initialization" approach to for explicit conversions (2);
- changes `index()` to return `-1` (now also known is `tuple_not_found`) if `empty()`;
- adds a visitation interface.

Beyond these requests, this revision

- discusses the options for relational operators, construction and assignments, with / from a same-type `variant`, an alternative, and a different `variant` type;
- hopefully makes the `variant` a regular type.

Differences to revision 2 (N4450)

- Everything requested by LEWG, most notably, `variant` now models a discriminated union
- `hash<variant<int>>` can now return different values than `hash<int>` (and it should - presumably it should take the index() into account).

- Describe template <size_t,...> get<I,...>(variant)
- Remove is_alternative that is not strictly needed to make variant usable (LEWG feedback).
- Remove std::swap() specialization; the default is just fine.
- Add obligatory introductory quote.
- Expanded on disadvantages of double buffering.

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 proposal 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-type-safe version of variant. boost::variant (3) and eggs::variant (4) are similar to this proposal. This proposal tries to merge the lessons from optional.

Recursive variant

Recursive variants are variants that (conceptually) have itself as one of the alternatives. There are good reasons to add support for a recursive variant; for instance to build AST nodes. There are also good reasons not to do so, and use unique_ptr<variant<...>> instead. A recursive variant can be implemented as an extension to variant, see for instance what is done for boost::variant.

This proposal does not contain support for recursive `variant`s; it also does not preclude a proposal for them.

What *is* supported by this proposal is a `variant` that has as one alternative a `variant` of a different type.

Visitor

Motivation

A good `variant` needs a visitor, multimethods, or any other dedicated access method. This proposal includes a visitor - not because it's the optimal design for accessing the elements, but because it *is* a design. Visitors are common, well understood and thus warrant inclusion in this proposal, independently of future, improved patterns.

Example

The content of a `variant` can thus be accessed as follows:

```
variant< ... > var = ...;
visit([](auto& val) { cout << val; }, []() {}, var);
```

using Lambda syntax; or

```
struct my_visitor {
  template <class AltType>
  ostream& operator()(AltType& var) { cout << var; return cout; }
};

variant< ... > var = ...;
visit(my_visitor(), var);
```

Return type of `visit()`

All `callable(T_i)` as well as `callable()` (depending on the `visit` overload used) must return the same type.

Visitor state

Overloads of the `visit` functions take non-`const` visitor callables, they allow functions to be invoked that change the state of the callable. Non-mutable lambdas on the other hand require overloads taking `const` callables.

Possible implementation characteristics

The closed set of types makes it possible to construct a constexpr array of functions (jump table) to call for each alternative. The visitation of a non-empty `variant` is then calling the array element at position `index()`, which is an `O(1)` operation.

Design considerations

A `variant` can be empty

To simplify the `variant` and make it conceptually composable for instance with `optional`, it is desirable that it always contains a value of one of its template type parameters. But the `variant` proposed here does have an empty state. Here is why.

The problem Here is an example of a state transition due to an assignment of a `variant` `w` to `v` of the same type:

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

In the last line, `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`.

Requiring `is_nothrow_copy_constructible` This problem exists only for a subset of types. These set of these types depends on implementations (implementations are free to add `noexcept`). Determining whether a type is `is_nothrow_copy_constructible` is not trivial, especially for novice users.

How does `union` do it? C++ `union`s get around this by not managing type transitions. Assignments involving type transitions are too desirable to forbid them.

Double-buffering An alternative used by `boost::variant` is to introduce a second buffer: `v` constructs the assigned value in this second buffer, leaving the previous value untouched. Once the construction of the new value was successful, the old value will be destructed and the `variant` flips type state and remembers that the current value is now stored in the secondary buffer. The disadvantages of a secondary buffer are

- additional, up to doubled memory usage: at least the largest alternative type for which `is_nothrow_copy_constructible` evaluates to `false` must fit in the secondary buffer (plus a boolean indicating which buffer is currently holding the object);
- the additional memory is - at least for `boost::variant` - allocated when needed in free store; it could also be stored wherever the `variant`'s storage is;
- the address of the `variant`'s internal storage changes;
- surprising sequencing of destruction and construction.

To demonstrate the latter, consider the following code:

```
struct X {
  X() { currentX = this; }
  ~X() { currentX = 0; }
  static X* currentX;
};
X* X::currentX = 0;

struct A: X { A() noexcept(false) {} };
struct B: X { B() noexcept(false) {} };

void sequencing() {
  // A double-buffered variant NOT proposed here:
  variant_db<A,B> v{A()};
  v.emplace<B>(); // copy-constructs B, then destroys A.
  assert(X::currentX && "surprising sequencing!"); // assert fails.
}
```

We believe that this is behavior is surprising; combined with the extra cost of the double buffer (at least in certain cases) we prefer other options.

Valid but unspecified, visibility of emptiness A `variant` can only become empty in exceptional cases: during an assignment or an emplacement, the copy or move constructor must throw. This limits the cases for which emptiness needs to be handled. An empty `variant` can be thought of as a moved-from `variant`: it is in a (perfectly) valid but unspecified state.

This state needs to be visible: accessing its contents or visiting it will violate preconditions; users must be able to verify that a `variant` is not in this state. Handling the exception from the assignment that creates an empty `variant` is not always possible; a `variant` passed to a function would become invalid, cannot be "healed" (all constructors / emplace etc of any alternative type might throw), and so the callee has no way of communicating to the outside that the variant is invalid.

Instead, we prefer to make this state visible through the `index()` returning `tuple_not_found` and a usability feature `empty()`. `empty()` is the name used for `any`, too (though not for `optional`, but this proposal cannot fix that).

Empty state and default construction Default construction of a `variant` should be allowed, to increase usability for instance in containers. LEWG opted against a `variant` default-initialized into its empty state, to limit the paths of getting an empty `variant` to a throwing construction during an assignment or call to `emplace()`.

Instead, the `variant` can be initialized with the first alternative (similar to the behavior of initialization of a `union`) if it is default constructible. For cases where this behavior should be explicit, and for cases where no such default constructible alternative exists, there is a separate type`empty_t` that can be used as first alternative, to explicitly enable default construction.

`constexpr` access

Many functions of `variant` can be marked `constexpr` without requiring "compiler magic" due to `reinterpret_cast`s of the internal buffer. This is strictly an extension of how `constexpr` can be implemented for the interfaces of `optional`; possible implementations involve recursive unions.

`noexcept`

The `variant` should ideally have the same `noexcept` clauses as `tuple`.

`variant<int, int>`

Multiple occurrences of identical types are allowed. They are distinct states; the `variant` can contain either the first or the second `int`. This models a discriminated union. For a `variant` with duplicate types in the alternatives, none of the interfaces that identify the alternative through a type template parameter (constructor, emplace, get, etc) can be used. Instead, `get<0>` has to be used in place of `get<int>`; emplacement-with-index instead of assignment or type-based emplacement; and construction providing an emplacement hint (`emplaced_index<0>`) instead of a construction passing an `int`.

`void` as an alternative

Again to facilitate meta-programming, `void` is an acceptable template type parameter for a `variant`. The `variant` will never store an object of this type, the position of the `void` alternative will never be returned by `index()`.

`variant<>`

A `variant` without alternatives cannot be constructed; it is otherwise an allowed type. It is easier to allow it than to forbid it.

`variant<int, const int>`

A `variant` can handle `const` types: they can only be set through `variant` construction and `emplace()`. If both `const` and non-`const` types are alternatives, the active alternative is chosen by regular constructor instantiation / overload rules, just as for any other possibly matching alternative types.

`variant<int&>`

References are supported as alternative types. Assignment to such a value is ill-formed.

Perfect Initialization

We employ the same mechanisms for perfect initialization (2) as `optional`; see the discussion there. A constructor tag `emplaced_type` is used to signal the perfect forwarding constructor overload.

Heterogenous and Element Assignment, Conversion and Relational Operators

This proposal thus follows the implementation of Boost.Variant and Eggs.Variant and only provides same-type relational operators. This is partially a consequence of the LEWG review, partially a requirement of `variant` being a regular type. As an example, transitivity can be violated if variants can be compared with their values:

```
variant_with_element_less<float, int> vi(12), vf(14.);
assert(
  vi < 13. < vf < vi &&
  R"quote(
```

```
    "Oh dear," says God, "I hadn't thought of that,"
    and promptly vanishes in a puff of logic.
)quote")
```

Assignment, Emplace

The assignment and emplace operations of `variant` model that of `optional`; also `variant` employs same-type optimizations, using the assignment operator instead of construction if the `variant` already contains a value of the assigned / emplaced type.

Access to List of Alternative Types

C++ does not currently define a way to specify a template parameter type list. This should be proposed. Until then, `variant` offers access through an undefined template class `type_list`, with `variant` instantiations providing a typedef to their template parameter list as `types`.

Access to Address of Storage

Given that `variant` is type safe, access to the address of its internal storage is not provided. If really needed, that address can be determined by using a visitor.

Feature Test

No header called `variant` exists; testing for this header's existence is thus sufficient.

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.

Changes to header `<tuple>`

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

```
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

```
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

```
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.?, Empty alternative
struct empty_t {};

// 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(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>
  const remove_reference_t<tuple_element_t<I, variant<Types...>>>*
```

```
    get(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... Types>
  decltype(auto) visit(Visitor&, const variant<Types...>&);

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

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

  template <class Visitor, class... Variants>
  decltype(auto) visit(const Visitor&, const tuple<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.? type list access
    template <class... > struct type_list;
    typedef type_list<Types...> types;

    // 2.? variant construction
    constexpr variant() noexcept(see below);
    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.?, 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, 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&&...);
    template <size_t I, class U, class... Args>
      void emplace(initializer_list<U>, Args&&...);

    // 2.?, value status
    bool empty() 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 has a contained object that is in an empty but unspecified 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. Implementations 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,sizeof...(Types))` in order, and `T_i` be the `i`th type in `Types`.

`constexpr variant() noexcept(see below)`

Effects: Constructs a `variant` holding a default constructed value of `T_0`.
Postconditions: `index()` is `0`.
Throws: Any exception thrown by the default constructor of `T_0`.
Remarks: The function shall not participate in overload resolution unless `is_default_constructible_v<T_0>` is `true`. The expression inside `noexcept` is equivalent to `is_nothrow_default_constructible_v<T_0>`.

`variant(const variant& w)`

Requires: `is_copy_constructible_v<T_i>` is `true` for all `i`.
Precondition: `w.empty()` must be `false`.
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()`.
Precondition: `w.empty()` must be `false`.
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 variant(const T& t)`

Requires: `T` occurs in `Types...` exactly once and `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 there is exactly one occurrence 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: `T` occurs in `Types...` exactly once and `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 there is exactly one occurrence 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` occurs in `Types...` exactly once and `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 not participate in overload resolution unless there is exactly one occurrence 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` occurs in `Types...` exactly once 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_v<T, initializer_list<U>&, Args&&...>` is `true` and unless there is exactly one occurrence 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: 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.

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

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

Assignment

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

Requires: is_copy_constructible_v<T_i> && is_copy_assignable_v<T_i> is true for all i.

Precondition: rhs.empty() must be false.

Effects: If `index() == rhs.index()`, calls `get<j>(*this) = get<j>(rhs)` with `j` being `index()`. Else destructs the current contained value of `*this`. 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: `index() == rhs.index()`

Exception safety: If an exception is thrown during the call to `T_i`'s copy constructor (with `i` being `rhs.index()`, `empty()` will be `true` and no copy assignment will take place; the `variant` will be in a valid but unspecified state. If 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; `index()` will be `i`.

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

Requires: `is_move_constructible_v<T_i> && is_move_assignable_v<T_i>` is `true` for all `i`.

Precondition: `rhs.empty()` must be `false`.

Effects: If `!empty() && index() == rhs.index()`, the move-assignment operator is called to set the contained object to `std::forward<T_j>(get<j>(rhs))` with `j` being `rhs.index()`. Else destructs the current contained value of `*this` if `empty()` is `false`, then initializes `*this` to hold the same alternative as `rhs` and initializes the contained value with `std::forward<T_j>(get<j>(rhs))`.

Returns: `*this`.

Remarks: The expression inside `noexcept` is equivalent to: `is_nothrow_move_assignable_v<T_i> && is_nothrow_move_constructible_v<T_i>` for all `i`.

Exception safety: If an exception is thrown during the call to `T_j`'s copy constructor (with `j` being `rhs.index()`), `empty()` will be `true` and no copy assignment will take place; the `variant` will be in a valid but unspecified state. If an exception is thrown during the call to `T_j`'s move assignment, the state of the contained value is as defined by the exception safety guarantee of `T_j`'s move assignment; `index()` will be `j`.

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

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

Effects: Destructs the currently contained value if `empty()` is `false`. 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, `empty()` will be `true`; the `variant` will be in a valid but unspecified state.

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

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

Effects: Destructs the currently contained value if empty() is false. 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, empty() will be true; the variant will be in a valid but unspecified state.

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

```
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 empty() is false. 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, empty() will be true; the variant will be in a valid but unspecified state.

```
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 empty() is false. 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, empty() will be true; the variant will be in a valid but unspecified state.

Remarks: The function shall not participate in overload resolution unless is_constructible_v<tuple_element<I, variant>, initializer_list<U>&, Args&&...> is true.

bool empty() const noexcept

Effects: returns whether the variant contains a value or is in a valid but unspecified state.

size_t index() const noexcept

Effects: Returns the index j of the currently active alternative, or tuple_not_found if empty() is true.

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

Requires: !empty() && !rhs.empty(). is_move_constructible_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 swap(get<i>(*this), get<i>(hrs)), the state of the value of this and of rhs is determined by the exception safety guarantee of swap for lvalues of T_i with i being index(). If an exception is thrown during the call to swap(*this, hrs), the state of the value of this and of rhs is determined by the exception safety guarantee of variant's move constructor and assignment operator.

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;
```

Template instances of emplaced_type_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_type_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.

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` does not equal to `index()`,
- for `get` overloads with template parameter list `class T, class... Types`, because `holds_alternative<T>(v)` is `false`

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

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`.

tuple interface to class template variant

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

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

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

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

Value access

`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...>>`.

`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...>&)`

Requires: The type `T` occurs exactly once in `Types...`. Otherwise, the program is ill-formed. `v.empty()` must be `false`.
Effects: Equivalent to `return get<tuple_find<T, variant<Types...>>::value>(v)`.
Throws: Any exceptions that the expression in the Effects clause throws.

`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. `v.empty()` must be `false`.
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)`. `v.empty()` must be `false`.

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)`. `v.empty()` must be `false`.

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. `v->empty()` must be `false`.

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). v.empty() must be false.

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: !empty() && !v.empty() shall be true. get<i>(v) == get<i>(w) is a valid expression returning a type that is convertible to bool, for for all i in 0 ... sizeof...(Types).

Returns: 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: !empty() && !v.empty() shall be true. get<i>(v) < get<i>(w) is a valid expression returning a type that is convertible to bool, for for all i in 0 ... sizeof...(Types).

Returns: 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...  Types> decltype(auto) visit(Visitor&
vis, const variant<Types...>& var)
```

```
template <class Visitor, class...  Types> decltype(auto) visit(const
Visitor& vis, const variant<Types...>& var)
```

Requires: var.empty() must be false. vis(get<i>(var)) must be a valid expression of the same type, for all i.

Effects: Calls vis(get<i>(var)) with i being var.index().

Remarks: The invocation of the callable must be implemented in O(1), i.e. it must must not depend on sizeof...(Types).

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

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

Requires: var.empty() must be false for all var in vars. 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().

Remarks: Unlike for the single variant visitor, 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, basic ingredients.

Acknowledgments

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

References

1. *Working Draft, Technical Specification on C++ Extensions for Library Fundamentals.* N4335

2. *Improving pair and tuple, revision 2.* N4064

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

4. *Eggs.Variant* [online]. Available from: http://eggs-cpp.github.io/variant/