Variant: a type-safe union that is rarely invalid (v5).

P0088R0, ISO/IEC JTC1 SC22 WG21

Axel Naumann (axel@cern.ch), 2015-09-27

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&) {}
```

It is a sibling of the proposal P0087 named "Variant: a type-safe union without undefined behavior (v2)". Background information and design discussion are available in P0086.

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

Results of the second LEWG review in Lenexa

- Name of empty state:
- empty: 0
- error: 6
- invalid: 14
- bad: 5
- fail: 0
- partially formed: 4
- Name of query function:
  - query function: valid 13
  - is_valid 2
  - invalid 1
  - is_invalid 2
  - explicit operator bool 7
  - index() == tuple_not_found 10

- Upon invalid, should index return a magic value? SF=5, F=3, N=1, A=2, SA=2
- index() has a precondition of being valid() (otherwise UB) SF=5 F=2 N=0 A=3 SA=3
- What do we want to call the “empty_t” stand-in type?
  - empty_t 4
  - empty 4
  - one_t 1
  - blank 6
  - blank_t 7
  - monostate 7

  Runoff:
  - blank* 3
  - monostate 8
- Add assignment from an exact type if the type is unique? Unanimous consent.
- Add an example of multi-visititation; change visit() to a variadic signature.
- Keep names in_place_type and in_place_index to be consistent with optional? General consent.

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 `!valid()`;
- 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 (N4516)

- 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.

Differences to revision 3 (N4450)

- Added discussion of (semi-) destructive move.
- Assignment from an alternative types are back.
- Multi-visitation example added.
- `visit()` is now variadic.
- Implemented several suggestions by Peter Dimov: removed `type_list`; reduced probability of `!valid()` for copy assignment / construction.
- Renamed to monostate, get_if().

Differences to revision 4 (N4542)

- Make `valid()` a visible state for value extraction functions (`get()`, `visit()`).
- Move general design discussion into P0086.
- Remove `valid()` precondition for copy / move construction from a `variant`.

Discussion

Additional empty state

LEWG opted against introducing an explicit additional variant state, representing its invalid (and possibly empty, default constructed) state. This is meant to simplify the `variant` use: as getting a `variant` into the invalid state is sufficiently difficult, it was felt that there is no need to regularly check for a variant becoming invalid. This prevents all `get<int>(v)` calls from being protected by `if (v.valid())`.

Visibility of the Invalid State

Accessing an invalid variant's value is undefined behavior, whatever alternative is accessed.

The `variant`'s invalid 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.

When in the invalid state, `index()` returns `tuple_not_found`; `variant` provides `valid()` as a usability feature.

This usually does not need to be checked given how rare the invalid case is. It (generally) keeps a variant with N alternatives as an N-state type.

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 invalid state, to make invalid `variant`s really rare.

Instead, the `variant` can be initialized with the first alternative (similar to the behavior of initialization of a `union`) only if that 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 `monostate` that can be used as first alternative, to explicitly enable default construction.

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 valid, 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.?, Explicitly default-constructed alternative
  struct monostate {};
  bool operator<(const monostate&, const monostate&) constexpr
    { return false; }
  bool operator>(const monostate&, const monostate&) constexpr
    { return false; }
  bool operator<=(const monostate&, const monostate&) constexpr
    { return true; }
  bool operator>=(const monostate&, const monostate&) constexpr
    { return true; }
  bool operator==(const monostate&, const monostate&) constexpr
    { return true; }
  bool operator!=(const monostate&, const monostate&) constexpr
    { return false; }

  // 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...>*);

template <size_t I, class... 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...>>;
  template <class... Types>
    struct hash<experimental::monostate>;
} // namespace std
```

Class template `variant`

```
namespace std {
namespace experimental {
inline namespace fundamentals_vXXXX {
  template <class... Types>
  class variant {
  public:

    // 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> variant& operator=(const T&);
    template <class T> variant& operator=(const T&&) noexcept(see below);

    // 2.?, `variant` modifiers
    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 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. 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`$^{\text{th}}$ 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 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<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.

Modifiers

`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<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.

```
template <size_t I, class U, class... Args> constexpr explicit
variant(emplaced_index_t<I>, initializer_list<U> 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: If valid() is true, 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.

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 false and no copy assignment will take place; the variant will be in a valid but partially 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.

Effects: If valid() && 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 valid() is true, 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 move constructor (with j being rhs.index()), valid() will be false and no move assignment will take place; the `variant` will be in a valid but partially 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> variant& operator=(const T& t)`

`template <class T> variant& operator=(const T&& t) noexcept(see below)`

Requires: The overload set `T_i(t)` of all constructors of all alternatives of this `variant` must resolve to exactly one best matching constructor call of an alternative type `T_j`, according to regular overload resolution; otherwise the program is ill-formed. [Note:

```
variant<string, string> v;
v = "abc";
```

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_constructibe<T_j>` is `true`, creates a temporary of type `T_j`, passing `t` as argument to the selected constructor. Destructs 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_constructibe<T_j>` is `true`, and copy-constructing the contained value passing `t` if `is_move_constructibe<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`.

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

Requires: `is_constructible_v<T, 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 `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&&...)

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`.

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.

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 `is_constructible_v<tuple_element<I, variant>, initializer_list<U>&, Args&&...>` 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 `tuple_not_found` if `valid()` is `false`.

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

Requires: `valid() && rhs.valid()`. `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.valid()` must be `true`.
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.valid() must be true.
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.valid() must be true.
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.valid() must be true.
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->valid() must be true.
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.valid()` must be `true`.

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: `valid() && v.valid()` 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: `valid() && v.valid()` 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... Variants>   decltype(auto)
visit(Visitor& vis, Variants&... vars)
```

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

Requires: `var.valid()` must be `true` 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: `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-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