Variant: a type-safe union (v4).

N4542, ISO/IEC JTC1 SC22 WG21

Axel Naumann (axel@cern.ch), 2015-05-24

Contents

Introduction 4

Version control 5
  Results of the LEWG review in Urbana 5
  Results of the LEWG review in Lenexa 5
  Results of the second LEWG review in Lenexa 7
  Differences to revision 1 (N4218) 8
  Differences to revision 2 (N4516) 8
  Differences to revision 3 (N4450) 9

Discussion 9
  A variant is not boost::any 9
  union versus variant 9
  Other implementations 9
  Recursive variant 10
  Visitor 10
    Motivation 10
    Example 10
    Return type of visit() 11
    Visitor state 11
    Possible implementation characteristics 11
Design considerations .............................................. 11
A variant can be empty ............................................ 11
constexpr access ................................................. 14
noexcept .......................................................... 14
variant<int, int> .................................................. 14
void as an alternative ............................................ 14
variant<> .......................................................... 15
variant<int, const int> ........................................... 15
variant<int&> ..................................................... 15
Perfect Initialization .............................................. 15
Heterogenous and Element Assignment, Conversion and Relational
Operators .......................................................... 15
Assignment, Emplace .............................................. 16
Access to Address of Storage .................................... 16
Feature Test ......................................................... 16

Variant Objects .................................................. 16
In general .......................................................... 16
Changes to header <tuple> ....................................... 16
Header <variant> synopsis ....................................... 17
Class template variant .......................................... 20
Construction ....................................................... 21
Destructor .......................................................... 24
Assignment .......................................................... 24
bool valid() const noexcept ...................................... 28
size_t index() const noexcept .................................... 28
void swap(variant& rhs) noexcept(see below) ................. 28
In-place construction ............................................ 28
class bad_variant_access ....................................... 29
  bad_variant_access(const string& what_arg) ............... 29
  bad_variant_access(const char* what_arg) ................. 29
tuple interface to class template variant .................... 30
template <class T, class... Types> struct tuple_size
    <variant<Types...>> ...................................... 30

template <size_t I, class... Types> struct tuple_element<I,
    variant<Types...>> ........................................ 30

Value access ................................................. 30

template <class T, class... Types> bool holds_alternative(const
    variant<Types...>& v) noexcept; .......................... 30

template <class T, class... Types> remove_reference_t<T>&
    get(variant<Types...>& v) ............................. 30

template <class T, class... Types> const remove_reference_t<T>&
    get(const variant<Types...>& v) ......................... 30

template <class T, class... Types> T&& get(variant<Types...>&&
    v) .......................................................... 30

template <size_t I, class... Types> remove_reference_t<T>&
    get(variant<Types...>& v) ............................. 31

template <size_t I, class... Types> const remove_reference_t<T>&
    get(const variant<Types...>& v) ......................... 31

template <size_t I, class... Types> T&& get(variant<Types...>&&
    v) .......................................................... 31

template <class T, class... Types> remove_reference_t<T>*
    get(variant<Types...>* v) ............................. 31

template <class T, class... Types> const remove_reference_t<T>*
    get(const variant<Types...>* v) ......................... 31

template <size_t I, class... Types> remove_reference_t<tuple_element_t<I,
    variant<Types...>>>* get(variant<Types...>*) .......... 32

template <size_t I, class... Types> const remove_reference_t<tuple_element_t<I,
    variant<Types...>>>* get(const variant<Types...>*) .... 32

Relational operators ....................................... 32

template <class... Types> bool operator==(const
    variant<Types...>& v, const variant<Types...>& w) 32

template <class... Types> bool operator!=(const
    variant<Types...>& v, const variant<Types...>& w) 32

template <class... Types> bool operator<(const
    variant<Types...>& v, const variant<Types...>& w) 32

template <class... Types> bool operator>(const
    variant<Types...>& v, const variant<Types...>& w) 32

template <class... Types> bool operator<(const
    variant<Types...>& v, const variant<Types...>& w) 32
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.
```
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 *`, `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().

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 `variants`; 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:

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

using Lambda syntax; or

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

variant<int, int, string> var{"abc"};
visit(my_visitor(), var);
```

Multi-visitation passes the current value of multiple variants to the visitor function. Example:
struct my_visitor {
    void operator()(...) { cout << "no match"; }
    void operator()(int i, double d, char c) {
        cout << i << ' ' << d << ' ' << c;
    }
};

variant<int, string> var1{12};
variant<long, double> var2{13};
variant<string, char> var3{'x'};
visit(my_visitor(), var1, var2, var3); // "12 13.0 x"

var2 = 42;
visit(my_visitor(), var1, var2, var3); // "no match"

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:

\[
\text{variant}<S, T> \; v = S(); \\
\text{variant}<S, T> \; w = T(); \\
v = w;
\]

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 \(-\text{variant}\), and it must make this state visible, because any call of \( \text{get}<T>(v) \) would access an invalid object. The most straight-forward option is to introduce a new, empty state of the \text{variant}.

Requiring \text{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 \text{noexcept}). Determining whether a type is \text{is_nothrow_copy_constructible} is not trivial, especially for novice users.

Changing \text{no throw} of move constructor  Some of constructors can throw, for instance standard library types implemented with sentinel nodes. One could however implement traits signaling these types as “once moved, the moved-from object cannot be assigned to”, also known as (semi-) destructive move. This would cover all known implementations of all standard library types; variants could contain all of them as alternatives.

But this still makes these types unusable for instance as data members of user structs. The member itself could be contained in a variant, because its trait specialization tell variant that it can be moved, noexcept, even if maybe only destructively moved. But the compound user type will still not be \text{noexcept(true)}; users would need to specialize the (semi-) destructive move trait for their type. This is far too complex.

How does union do it?  \text{C++ unions} get around this by not managing type transitions. Assignments involving type transitions are too desirable to forbid them.

Double-buffering  An alternative used by \text{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 \text{variant} flips type state and remembers that the current value is now stored in the secondary buffer. The disadvantages of a secondary buffer are
Discussion

- 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:

```cpp
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 \texttt{index()} returning \texttt{tuple\_not\_found} and a usability feature \texttt{valid()}.

**Empty state and default construction**  Default construction of a \texttt{variant} should be allowed, to increase usability for instance in containers. LEWG opted against a \texttt{variant} default-initialized into its empty state, to limit the paths of getting an empty \texttt{variant} to a throwing construction during an assignment or call to \texttt{emplace()}. Instead, the \texttt{variant} can be initialized with the first alternative (similar to the behavior of initialization of a \texttt{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 \texttt{monostate} that can be used as first alternative, to explicitly enable default construction.

\texttt{constexpr} access

Many functions of \texttt{variant} can be marked \texttt{constexpr} without requiring “compiler magic” due to \texttt{reinterpret\_casts} of the internal buffer. This is strictly an extension of how \texttt{constexpr} can be implemented for the interfaces of \texttt{optional}; possible implementations involve recursive unions.

\texttt{noexcept}

The \texttt{variant} should ideally have the same \texttt{noexcept} clauses as \texttt{tuple}.

\texttt{variant<int, int>}

Multiple occurrences of identical types are allowed. They are distinct states; the \texttt{variant} can contain either the first or the second \texttt{int}. This models a discriminated union. For a \texttt{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, \texttt{get<0>} has to be used in place of \texttt{get<int>}; emplacement-with-index instead of assignment or type-based emplacement; and construction providing an emplacement hint (\texttt{emplaced\_index<0>}) instead of a construction passing an \texttt{int}.

\texttt{void} as an alternative

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

\textit{N4542} Discussion
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:

```cpp
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")
```

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

Specific care was taken in the copy-assignment operator and copy constructor to reduce the likelihood of the variant ending up in an invalid state: whenever possible, they will first create a temporary, and only destruct the contained type if this temporary construction did not throw. The variant will thus only end up in an invalid state if the move constructor throws (which is far less likely even if they are `noexcept(false)`), or if the alternative cannot be move constructed.

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 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:

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

\begin{verbatim}
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>> {};

tuple_find<T, variant<Types...>>:
  public tuple_find<T, tuple<Types...>> {};
\end{verbatim}

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

\textbf{Header <variant> synopsis}

\begin{verbatim}
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; }
\end{verbatim}
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 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;

} // namespace std
Class template `variant`

```cpp
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, 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);

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&&...);

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

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

\texttt{constexpr \texttt{variant}() noexcept(see below)}

**Effects:** Constructs a \texttt{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 \texttt{noexcept} is equivalent to \texttt{is_nothrow_default_constructible_v<T_0>}. The function shall be \texttt{= delete} if \texttt{is_default_constructible_v<T_0>} is false.

\texttt{\texttt{variant}(\texttt{const \texttt{variant}& w})}

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

**Precondition:** \(w\).valid() must be true.

**Effects:** initializes the \texttt{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 \(i\).

\texttt{\texttt{variant}(\texttt{\texttt{variant}&& w}) noexcept(see below)}

**Requires:** \texttt{is_move_constructible_v<T_i>} is true for all \(i\).

**Effects:** initializes the \texttt{variant} to hold the same alternative as \(w\). Initializes the contained value with \texttt{std::forward<T_j>(get<j>(w))} with \(j\) being \(w\).index().

**Precondition:** \(w\).valid() must be true.

**Throws:** Any exception thrown by the selected constructor of any \(T_i\) for all \(i\).

**Remarks:** The expression inside \texttt{noexcept} is equivalent to the logical AND of \texttt{is_nothrow_move_constructible<T_i>::value} for all \(i\).

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

**Requires:** \texttt{is_copy_constructible_v<T>} is true.

**Effects:** initializes the \texttt{variant} to hold the alternative \(T\). Initializes the contained value to a copy of \(t\).

**Postconditions:** \texttt{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<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.
Precondition: rhs.valid() must be true.
N4542 Variant Objects

Effects: If \( \text{index()} == \text{rhs.index()} \), calls \( \text{get<j>}(\ast\text{this}) = \text{get<j>}(\text{rhs}) \) with \( j \) being \( \text{index()} \). Else copies the value contained in \( \text{rhs} \) to a temporary, then destroys the current contained value of \( \*\text{this} \). Sets \( \*\text{this} \) to contain the same type as \( \text{rhs} \) and move-constructs the contained value from the temporary.

Returns: \( \*\text{this} \).

Postconditions: \( \text{index()} == \text{rhs.index()} \)

Exception safety: If an exception is thrown during the call to \( T_i \)'s copy constructor (with \( i \) being \( \text{rhs.index()} \)), \( \*\text{this} \) will remain unchanged. If an exception is thrown during the call to \( T_i \)'s move constructor, \( \text{valid()} \) will be \( \text{false} \) and no copy assignment will take place; the \( \text{variant} \) 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; \( \text{index()} \) will be \( i \).

\[
\text{variant\& operator=}(\text{const variant\&\& rhs}) \text{ noexcept}(\text{see below})
\]

Requires: \( \text{is\_move\_constructible}\_v<T_i> \&\& \text{is\_move\_assignable}\_v<T_i> \) is true for all \( i \).

Precondition: \( \text{rhs.valid()} \) must be true.

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

Returns: \( \*\text{this} \).

Remarks: The expression inside \text{noexcept} is equivalent to: \( \text{is\_nothrow\_move\_assignable}\_v<T_i> \&\& \text{is\_nothrow\_move\_constructible}\_v<T_i> \) for all \( i \).

Exception safety: If an exception is thrown during the call to \( T_j \)'s copy constructor (with \( j \) being \( \text{rhs.index()} \)), \( \text{valid()} \) will be \( \text{false} \) and no copy assignment will take place; the \( \text{variant} \) 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; \( \text{index()} \) will be \( j \).

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

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

Requires: The overload set \( T_i(t) \) of all constructors of all alternatives of this \text{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:]

```cpp
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_constructible<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_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:

```cpp
is_nothrow_move_assignable<T_i>::value && is_nothrow_move_constructible<T_i>::value
```

for all $i$.

```cpp
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 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 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 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 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.

```cpp
bool valid() const noexcept
```

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

```cpp
size_t index() const noexcept
```

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

```cpp
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 values 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

```cpp
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.

```cpp
class bad_variant_access
{
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).
N4542 Variant Objects

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 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-baloo Bergé and Antony Polukhin provided very valuable feedback, criticism and suggestions. Thanks also to Vincenzo Innocente and Philippe Canal for their comments.

References

2. Improving pair and tuple, revision 2. N4064