Variant: a typesafe union (v2).

N4450, ISO/IEC JTC1 SC22 WG21

Axel Naumann (axel@cern.ch), 2015-04-13

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>4</td>
</tr>
<tr>
<td>Version control</td>
<td>5</td>
</tr>
<tr>
<td>Results of the LEWG review in Urbana</td>
<td>5</td>
</tr>
<tr>
<td>Differences to revision 1 (N4218)</td>
<td>5</td>
</tr>
<tr>
<td>Discussion</td>
<td>6</td>
</tr>
<tr>
<td>A variant is not boost::any</td>
<td>6</td>
</tr>
<tr>
<td>union versus variant</td>
<td>6</td>
</tr>
<tr>
<td>Other implementations</td>
<td>6</td>
</tr>
<tr>
<td>Recursive variant</td>
<td>6</td>
</tr>
<tr>
<td>Visitor</td>
<td>7</td>
</tr>
<tr>
<td>Motivation</td>
<td>7</td>
</tr>
<tr>
<td>Visiting an empty variant</td>
<td>7</td>
</tr>
<tr>
<td>Example</td>
<td>7</td>
</tr>
<tr>
<td>Return type of visit()</td>
<td>7</td>
</tr>
<tr>
<td>Visitor state</td>
<td>8</td>
</tr>
<tr>
<td>Possible implementation characteristics</td>
<td>8</td>
</tr>
<tr>
<td>Design considerations</td>
<td>8</td>
</tr>
<tr>
<td>A variant can be empty</td>
<td>8</td>
</tr>
<tr>
<td>constexpr access</td>
<td>10</td>
</tr>
</tbody>
</table>
N4450

Contents

noexcept ........................................... 10

variant<int, int> ................................. 11

void as an alternative ............................ 11

variant<int, const int> ......................... 11

Perfect Initialization ............................. 11

Assignment, Conversion, Relational Operators .......... 11

Variant Objects .................................. 12

In general ........................................... 12

Changes to header <tuple> .......................... 13

Header <variant> synopsis ......................... 13

Class template variant ............................ 15

Construction ....................................... 17

Destructor .......................................... 20

Assignment ......................................... 20

void clear() ....................................... 24

bool empty() const noexcept ....................... 24

size_t index() const ................................ 24

void swap(variant& rhs) noexcept; //see below ...... 24

In-place construction .............................. 24

class bad_variant_access ......................... 25

bad_variant_access(const string& what_arg) .......... 25

bad_variant_access(const char* what_arg) .......... 25

tuple interface to class template variant .......... 25

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

<variant<Types...>> ............................... 25

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

variant<Types...>> ............................... 25

Value access ....................................... 25

template <class T, class... Types> constexpr bool

is_alternative(const variant<Types...>& v)

noexcept ........................................... 25

2
template <class T, class... Types> bool holds_alternative(const variant<Types...>& v) noexcept; .................. 26

template <class T, class... Types> T& get(variant<Types...>& v) ........................................ 26

template <class T, class... Types> T&& get(variant<Types...>&& v) ....................................... 26

template <class T, class... Types> const T& get(const variant<Types...>&) ............................... 26

Relational operators .................................................. 26

template <class... Types> bool operator==(const variant<Types...>& v, const variant<Types...>& w) 26

template <class... Types> bool operator!=(const variant<Types...>& v, const variant<Types...>& w) 27

template <class... Types> bool operator<(const variant<Types...>& v, const variant<Types...>& w) 27

template <class... Types> bool operator>(const variant<Types...>& v, const variant<Types...>& w) 27

template <class T, class... Types> bool operator==(const variant<Types...>& v, const T& t) .............. 28

template <class T, class... Types> bool operator!=(const variant<Types...>& v, const T& t) .............. 28

template <class T, class... Types> bool operator<(const variant<Types...>& v, const T& t) .............. 28

template <class T, class... Types> bool operator>(const variant<Types...>& v, const T& t) .............. 28

Specialized algorithms ............................................... 29
Introduction

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

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

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

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

Differences to revision 1 (N4218)

As requested by the LEWG review in Urbana, this revision

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

Beyond these requests, this revision

- discusses the options for relational operators, construction and assignments, with / from a same-type variant, an alternative, and a different variant type;
- hopefully makes the variant a regular type.
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-typesafe version of variant. boost::variant (3) is very similar to this proposal. This proposal tries to merge the lessons from optional. It does not address the visitation pattern that is an integral part of the boost::variant, nor any special treatment of a recursive variant. This proposal is extremely similar to eggs::variant (4).

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 instead 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.

Visiting an empty variant

Because the variant might be empty, a no-value case must be available, too. There are two visit() overloads, one taking an object callable and a variant. On the callable object, both callable(T_i) (corresponding to get<T_i>, for all alternatives) and callable() (for the empty case) must be available. The second overload takes objects, callable and callable_empty, and a variant. On the callable object, callable(T_i) (corresponding to get<T_i>, for all alternatives) must be available. On the callable_empty object, callable_empty() (for the empty case) must be available.

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; }
    ostream& operator()() { return cout; }
};
```

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

Return type of visit()

The function called in the empty variant case has an important additional role: it defines the return type of all callable invocations. All callable(T_i) as well as either callable() or callable_empty() (depending on the visit overload used) must return the same type.
Visitor state

The single-callable overload takes a non-const reference to the callable: it allows functions to be invoked that change the state of the callable. This is as an additional motivation to have two overloads.

Possible implementation characteristics

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

Design considerations

A variant can be empty

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

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

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

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

Requiring is_nothrow_copy_constructible  This problem exists only for a subset of types; those that are is_nothrow_copy_constructible are guaranteed to not throw during the above assignment. Many types can not guarantee is_nothrow_copy_constructible, for instance because an allocation during copy construction might throw. One of the most common types that is not is_nothrow_copy_constructible is vector and classes that contain it. There are three options in the area of restricting assignment with non-is_nothrow_copy_constructible types:
1) Limit the alternatives to \texttt{is\_nothrow\_copy\_constructible} types. This is deemed a restriction too strong for a usable \texttt{variant}, especially as the motivation for this restriction is non-obvious (effect on “teachability”).

2) All alternatives must be \texttt{is\_nothrow\_copy\_constructible} or the assignment is illformed. This will be a surprise to most users; we prefer uniform behavior for all alternative types if at all possible.

3) Throw when a non-\texttt{is\_nothrow\_copy\_constructible} type is assigned, keeping the \texttt{variant}’s value unchanged. This is even worse and impossible to teach.

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

\textbf{Double-buffering} An alternative used by \texttt{boost::variant} is to introduce a second buffer: \texttt{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 \texttt{variant} flips type state and remembers that the current value is now stored in the secondary buffer. The disadvantages of a secondary buffer are

- additional, up to doubled memory usage: at least the largest alternative type for which \texttt{is\_nothrow\_copy\_constructible} evaluates to \texttt{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 \texttt{boost::variant} - allocated when needed in free store; it could also be stored wherever the \texttt{variant}’s storage is;
- surprising sequencing of destruction and construction.

To demonstrate the latter, consider the following code:

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

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

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 && "suprising sequencing!"); // assert fails.
}

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

**Making emptiness part of the type** Some existing implementations (5)
suggest the use of a template type parameter of a dedicated tag type, whose
presence in the list of template parameters signals that the variant is allowed to
be empty.

This adds complexity to an otherwise simple type. We believe that offering
a non-empty variant might be a viable future extension but that the basic
implementation should be simple and useful.

We are notably against (mis-)using a type template parameter of the `variant`
to signal whether the `variant` is allowed to be empty, as it overlays two concepts:
content versus personality. Instead we suggest to use a different template name.

**Empty state and default construction** If a `variant` were not allowed to
be empty, default construction becomes all but obvious: it could be disallowed
(making it much more difficult to use, for instance in containers) or it could be
initialized with the first default constructible alternative. The latter will likely
surprise most users and is non-obvious and outright dangerous behavior if that
default constructor has side-effects.

It is highly desirable for a `variant` to be default constructable (and non-
surprisingly so). Allowing for an empty state is the perfect solution.

**constexpr access**

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

**noexcept**

The `variant` should have the same `noexcept` clauses as `tuple`. 
N4450 Discussion

**variant<int, int>**

Multiple occurrences of identical types are allowed. This facilitates meta-programming; there is no need to find the unique set, even though the state query will treat the template parameter list as a set: it will return the smallest index of a type. I.e. in the example of variant<int, int>, index() on a non-empty variant returns 0 in all cases.

**void as an alternative**

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

**variant<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.

A variant holding a const object can be cleared and swapped.

**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.

**Assignment, Conversion, Relational Operators**

**Available overloads** The assignment operators as well as the relational operators suffer from an ambiguity: should the variant be assigned / compared, or a variant of different alternatives (subset, superset, unrelated and possibly convertible), or a value of an alternative? These different cases can be resolved partially through overload resolution enabling all of the following assignments:

```cpp
variant<int, float> v, w;
variant<int, double> x;
v = 42; // assign an alternative's value
w = v; // assign a variant of same type
x = v; // heterogeneous variant assignment
```
The overload template `<class T> variant operator=(T&&)` requires `T` to be one of the alternatives to participate in overload resolution; it would otherwise be used instead of the other overloads in all cases.

**Heterogeneous variant operations**  We propose to support the construction and assignment of variants of different type, only if all possible combinations of alternatives are well formed. Assignment can be possible in one direction but not in the other:

```cpp
variant<int, float> vif(42);
variant<int, void*> viv;
viv = vif; // well-formed; both int and float can be assigned to int
vif = viv; // ill-formed; void* value cannot be assigned to vif
```

Another option would be to delay these checks until runtime, and raise an exception if the current type combination does not allow the operation. We find this so undesirable that we decided against that option.

**Relational operators across alternatives**  Both Boost and Eggs take the alternative index into account in comparisons, which yields surprising results:

```cpp
variant<float, int> v1(12.f);
variant<float, int> v2(11);
v1 < v2 // true
```

The comparison returns `true` because the active index is different for `v1` and `v2`, and that of `v1` is smaller than that of `v2`. We find `false` as returned by the operators in this proposal a more obvious result in this case.

But this would require all pairs of alternatives to be comparable, which is a massive usability limitation; or the relational operators to throw if the comparison cannot be done, which is not a desirable behavior; or a hybrid comparison (compare values if possible, else alternative indices) which is inconsistent behavior and causes issues with usability and teachability.

This proposal thus follows the implementation of Boost.Variant and Eggs.Variant and only provides same-type relational operators.

**Variant Objects**

**In general**

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

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

```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...>>;
```

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

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

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

Header <variant> synopsis

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

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

template <size_t I, class... Types>
  tuple_element<I, variant<Types...>>::type&
      get(variant<Types...>&);
template <size_t I, class... Types>
  tuple_element<I, variant<Types...>>::type&&
      get(variant<Types...>&&);
template <size_t I, class... Types>
  const tuple_element<I, variant<Types...>>::type&
      get(const variant<Types...>&);

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

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

// 2.?, Visitation
template <class Visitor, class... Types>
decltype(auto) visit(Visitor&, const variant<Types...>&);

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

} // namespace fundamentals_vXXXX
} // namespace experimental

// 2.?, Hash support
template <class T> struct hash;
template <class... Types>
struct hash<experimental::variant<Types...>>;

} // namespace std

Class template variant

namespace std {
namespace experimental {
inline namespace fundamentals_vXXXX {

template <class... Types>
class variant {
    // 2.? type list access
    template <class... > struct type_list;
typedef type_list<Types...> types;

} // namespace fundamentals_vXXXX
} // namespace experimental
} // namespace std

} // namespace experimental


// 2.? variant construction
constexpr variant() noexcept;
variant(const variant&);
variant(variant&&) noexcept(see below);

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

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

template <class... UTypes> variant(const variant<UTypes...>&);
template <class... UTypes> variant(variant<UTypes...>&&);

// 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... UTypes> variant&
operator=(const variant<UTypes...>&);
template <class... UTypes> variant&
operator=(variant<UTypes...>&&);

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&&...);
void clear();

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

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

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

namespace fundamentals_vXXXX
}
} // namespace experimental
} // namespace std

Construction

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

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

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

constexpr variant() noexcept

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

variant(const variant& w)

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

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

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

Requires: is_alternative<T>(variant()) is true and is_copy_constructible<T>::value
  is true.
Effects: initializes the variant to hold the alternative T. Initializes the con-
tained value to a copy of t.
Postconditions: holds_alternative<T>(*this) is true
Throws: Any exception thrown by the selected constructor of T.
Remarks: If T's selected constructor is a constexpr constructor, this construc-
tor shall be a constexpr constructor.

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

Requires: is_alternative<T>(variant()) is true and is_move_constructible<T>::value
  is true.
Effects: initializes the variant to hold the alternative T. Initializes the con-
tained value with std::forward<T>(t).
Postconditions: holds_alternative<T>(*this) is true
Throws: Any exception thrown by the selected constructor of T.
Remarks: If T's selected constructor is a constexpr constructor, this construc-
tor shall be a constexpr constructor.

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

Requires: is_alternative<T>(variant()) is true and is_constructible<T,
  Args&&...>::value is true.
Effects: Initializes the contained value as if constructing an object of type T with the arguments std::forward<Args>(args)....
Postcondition: holds_alternative<T>(*this) is true
Throws: Any exception thrown by the selected constructor of T.
Remarks: If T's selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

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

Requires: is_alternative<T>(variant()) is true and is_constructible<T, initializer_list<U>&, Args&&...>::value is true.
Effects: Initializes the contained value as if constructing an object of type T with the arguments il, std::forward<Args>(args)....
Postcondition: holds_alternative<T>(*this) is true
Remarks: The function shall not participate in overload resolution unless is_constructible<T, initializer_list<U>&, Args&&...>::value is true.
Remarks: If T's selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

template <class... UTypes> explicit variant(const variant<UTypes...>& u);

Requires: is_copy_constructible<T_i>::value is true for all T_i that are part of Types and UTypes.
Effects: initializes the variant to hold the same alternative as u, or to an empty state if u was empty. If non-empty, initializes the contained value to a copy of the value contained by u.
Throws: Any exception thrown by the selected constructor of any T_i for all i. Throws bad_variant_access if the alternative held by u cannot be stored in *this.

template <class... UTypes> explicit variant(variant<UTypes...>&& u);

Requires: is_move_constructible<T_i>::value is true for all T_i that are part of Types... and UTypes....
Effects: initializes the variant to hold the same alternative as w. Initializes the contained value with std::forward<T_j>(get<j>(u)) with j being u.index().
Postconditions: empty() && u.empty() || holds_alternative<T_j>(*this) == holds_alternative<T_j>(u) is true with j being u.index().
**Throws**: Any exception thrown by the selected constructor of any \( T_i \) for all \( i \). Throws `bad_variant_access` if the alternative held by \( u \) cannot be stored in `*this`.

**Remarks**: The expression inside `noexcept` is equivalent to the logical AND of `is_nothrow_move_constructible<T_i>::value` for all types that are both in `Types...` and `UTypes...`.

**Destructor**

`~variant()`

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

**Assignment**

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

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

**Effects**: If `!this->empty() && this->index() == rhs.index()`, calls `get<j>(*this) = (get<j>(rhs))` with \( j \) being `this->index()`. Else destructs the current contained value of `*this` if `empty()` is `false`. Initializes `*this` to hold the same alternative as `rhs`. Initializes the contained value to a copy of the value contained by `rhs`.

**Returns**: `*this`.

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

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

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

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

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

**Returns:** \*this.

**Remarks:** The expression inside \texttt{noexcept} is equivalent to:

\begin{align*}
\text{is_nothrow_move_assignable\langle T_i \rangle::value} & \quad & \text{is_nothrow_move_constructible\langle T_i \rangle::value} \\
\text{for all } i.
\end{align*}

**Postconditions:** : \texttt{this->index()} == \texttt{rhs.index()}  
**Exception safety:**  
: If \texttt{rhs.empty()} is false and an exception is thrown during the call to \texttt{T_{-j}’s copy constructor} (with \texttt{j} being \texttt{rhs.index()}), \texttt{this->empty()} will be true and no copy assignment will take place. If \texttt{rhs.empty()} is false and an exception is thrown during the call to \texttt{T_{-j}’s copy assignment}, the state of the contained value is as defined by the exception safety guarantee of \texttt{T_{-j}’s copy assignment}; \texttt{this->index()} will be \texttt{j}.

**Example:**

\begin{verbatim}
variant<int, void*> viv(42);
variant<float, int> vfi;
vfi = viv; // ill-formed: void* cannot be assigned
\end{verbatim}
variant<int, float> vif(43);
variant<float, double> vfd;
vfd = vif; // ill-formed: assignment of int is ambiguous

variant<int, float> vif(43);
variant<float, string> vfs;
vfs = vif; // well-formed; get<float>(vfs) == 42.

Returns: *this.
Postconditions: this->index() == rhs.index()

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

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

Requires: The overload set of all constructors taking t of all alternatives of this variant must resolve to exactly one best matching constructor call, according to regular overload resolution.

Effects: If *this holds a T, the move-assignment operator is called to set the contained object to t. Else destructs the current contained value of *this, initializes *this to hold an alternative of the type selected by constructor overload resolution, and initializes the contained value by calling the selected constructor overload, passing t.

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

Exception safety: If an exception is thrown during the call to T's move constructor, this->empty() will be true and no copy assignment will take place. If an exception is thrown during the call to T's move assignment, the state of the contained value and t are as defined by the exception safety guarantee of T's copy assignment; this->index() will be j.

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

Requires: Shall not participate in overload resolution unless T is one of the alternative types.

Effects: If *this holds a T, the move-assignment operator is called to set the contained object to t. Else destructs the current contained value of *this
if empty() is false and initializes *this to hold the alternative T and initializes the contained value with std::forward<T>(t).

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

\[
\text{is_nothrow_move_assignable}<\text{T}_i>::\text{value} \&\& \text{is_nothrow_move_constructible}<\text{T}_i>::\text{value}
\]

for all i. Postcondition: : holds_alternative<T>(*this) is true. Exception safety: : If an exception is thrown during the call to T's move constructor, this->empty() will be true and no copy assignment will take place. If an exception is thrown during the call to T's move assignment, the state of the contained value and t's contained value are as defined by the exception safety guarantee of T_i's copy assignment; this->index() will be j.

\[
\text{template <class T, class... Args> void emplace(Args&&...)}
\]

Requires: is_constructible<T, Args&&...>::value is true.
Effects: Destructs the currently contained value. Then initializes the contained value as if constructing a value of type T with the arguments std::forward<Args>(args)....
Postcondition: holds_alternative<T>(*this) is true.
Throws: Any exception thrown by the selected constructor of T.

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

\[
\text{template <class T, class U, class... Args> void emplace(initializer_list<U> li, Args&&...)}
\]

Requires: is_constructible<T, initializer_list<U>&, Args&&...>::value is true.
Effects: Destructs the currently contained value. Then initializes the contained value as if constructing an object of type T with the arguments il, std::forward<Args>(args)....
Postcondition: holds_alternative<T>(*this) is true
Throws: Any exception thrown by the selected constructor of T.
Exception safety: If an exception is thrown during the call to T's constructor, this->empty() will be true.
Remarks: The function shall not participate in overload resolution unless
is_constructible<T, initializer_list<U>&, Args&&...>::value is true.
void clear()

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

bool empty() const noexcept

Effects: returns whether the variant contains a value.

size_t index() const

Effects: Returns the index $j$ of the first match of the contained value’s type $T_j$ in the variant’s template parameter list, or -1 if empty() is true.

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

Requires: LValues in $T_i$ shall be swappable and is_move_constructible<T_i>::value is true for all $i$.
Effects: calls swap(get<T_j>(), rhs->get<T_j>(rhs)) with $j$ being this->index().
Throws: Any exceptions that the expression in the Effects clause throws, including bad_variant_access.
Exception safety: If an exception is thrown during the call to function swap the state of the value of this and of rhs is determined by the exception safety guarantee of swap for lvalues of $T_j$ with $j$ being this->index(). If an exception is thrown during the call to $T_j$’s move constructor, the state of the value of this and of rhs is determined by the exception safety guarantee of $T_j$’s move constructor.

In-place construction

template <class T> struct emplaced_type_t{}
template <class T> constexpr emplaced_type_t<T> emplaced_type{};

Template instances of emplaced_type_t are empty structure types used as unique types to disambiguate constructor and function overloading, and signalling (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 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 \(\text{size}_t\ I, \text{class}...\ \text{Types}\), because \(I\) is out of range, * for **get** overloads with template parameter list \(\text{class} T, \text{class}...\ \text{Types}\), because \(\text{is_alternative}<T>(v)\) is false

**bad_variant_access(const string& what_arg)**

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

**bad_variant_access(const char* what_arg)**

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

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...>>

See the corresponding **tuple** specialization.

Value access

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

**Effects:** returns **true** if \(T\) is one of the alternatives of \(v\); **false** otherwise.
template <class T, class... Types> bool holds_alternative(const variant<Types...>& v) noexcept;

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

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

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

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

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

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

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

Relational operators

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

Requirements: 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.empty() && w.empty(); false if v.empty() != w.empty(). In all other cases, 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: 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: false if v.empty() && w.empty(); in all other cases, 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)

Requires: 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: false if v.empty() && w.empty(); in all other cases, 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... TTTypes, class... UTypes> bool operator<=(const variant<TTTypes...>& v, const variant<UTypes...>& w)

Returns: !(v > w).

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

Returns: !(v < w)
### Variant Objects

**Template**

```cpp
template <class T, class... Types> bool operator==(const variant<Types...>& v, const T& t)
```

*Requires:* \( \text{get<}i\text{>(v) == t} \) is a valid expression returning a type that is convertible to bool, for all \( i \) in \( 0 \ldots \text{sizeof...}(\text{Types}) \).

*Returns:* \( \text{get<j>(v) == t} \) with \( j \) being \( v\text{.index()} \).

**Template**

```cpp
template <class T, class... Types> bool operator!=(const variant<Types...>& v, const T& t)
```

*Returns:* \( !(v == t) \)

**Template**

```cpp
template <class T, class... Types> bool operator<(const variant<Types...>& v, const T& t)
```

*Requires:* \( \text{get<}i\text{>(v) < t} \) is a valid expression returning a type that is convertible to bool, for all \( i \) in \( 0 \ldots \text{sizeof...}(\text{Types}) \).

*Returns:* \( \text{get<j>(v) < t} \) with \( j \) being \( v\text{.index()} \).

**Template**

```cpp
template <class T, class... Types> bool operator>(const variant<Types...>& v, const T& t)
```

*Requires:* \( \text{get<}i\text{>(v) > t} \) is a valid expression returning a type that is convertible to bool, for all \( i \) in \( 0 \ldots \text{sizeof...}(\text{Types}) \).

*Returns:* \( \text{get<j>(v) > t} \) with \( j \) being \( v\text{.index()} \).

**Template**

```cpp
template <class T, class... Types> bool operator<=(const variant<Types...>& v, const T& t)
```

*Returns:* \( !(v > t) \)

**Template**

```cpp
template <class T, class... Types> bool operator>=(const variant<Types...>& v, const T& t)
```

*Returns:* \( !(v < t) \)
Specialized algorithms

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

Effects: calls \(x.\text{swap}(y)\).

Hash support

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

Requires: the template specialization \(\text{hash}\langle T_i \rangle\) shall meet the requirements of class template \(\text{hash}\) (C++11 §20.8.12) for all \(i\).

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

Visitation

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

Requires: \(\text{vis}()\) must be a valid expression. \(\text{vis}(\text{get}<i>(\text{var}))\) must be a valid expression returning the same type as \(\text{vis}()\), for all \(i\).

Effects: Calls \(\text{vis}()\) if \(\text{var.\text{empty}()}\) else \(\text{vis}(\text{get}<i>(\text{var}))\) with \(i\) being \(\text{var.\text{index}()}\).

\[
\text{template <class Visitor, class VisitorEmpty, class... Types> decltype(auto) visit(const Visitor& vis, const VisitorEmpty& vis_empty, const variant<Types...>& var)}
\]

Requires: \(\text{vis_empty}()\) must be a valid expression. \(\text{vis}(\text{get}<i>(\text{var}))\) must be a valid expression returning the same type as \(\text{vis_empty}()\), for all \(i\).

Effects: Calls \(\text{vis_empty}()\) if \(\text{var.\text{empty}()}\) else \(\text{vis}(\text{get}<i>(\text{var}))\) with \(i\) being \(\text{var.\text{index}()}\).

Conclusion

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

Thank you, Nevin “:-)" Liber, for bringing sanity to this proposal. Agustín K-baló 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