Variant design review.

P0086R0, ISO/IEC JTC1 SC22 WG21

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

Contents

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

Introduction

C++ needs a type-safe union. Given the wide variety of implementations and expectations, and given that two proposals by the same author are available, a combined review of the design choices seemed helpful. This paper contains background information on the proposals P0087 "Variant: a type-safe union without undefined behavior (v2)", and P0088 "Variant: a type-safe union that is rarely invalid (v5)", two different incarnations of a `variant`.

Context

A `variant` is not `std::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`

The proposals do not want 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` (1) and `eggs::variant` (2) are similar to this proposal. Their features are discussed in the relevant sections on design discussion.

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 to instead use `unique_ptr<variant<...>>` as an alternative. A recursive `variant` can be implemented as an extension to `variant`, see for instance what is done for `boost::variant`. The proposals does not contain support for recursive `variant`s; they also do not preclude a proposal for them.

What *is* supported by the proposals 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. The proposals include 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 the proposal, independently of future, improved patterns.

Example

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

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

using Lambda syntax; or

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

  ostream& operator()(const string& s) {
    cout << '"' << s << '"'; 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)` must return the same type to prevent possibly unexpected casts. Future versions can loosen this restriction if it is deemed too constraining.

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

Goals

This paragraph has obvious content, yet these goals might get out of sight in the following discussion. Jeffrey Yaskin has conducted a little unauthoritative survey with members of the committee (LEWG, LWG, EWG); the goals mentioned below are trying to summarize the outcome of that and the reflector discussions. Several goals are contradicting.

- Simplicity: a `variant` should be simple to understand for simple use cases. Its behavior shall not be surprising.
- Performance: if a `variant` has extra runtime cost, be it CPU cycles or memory, `union` will remain the type of choice for many. Keeping a `variant` to the size of the largest alternative size plus a tag, and keeping the operations to the fastest possible (for instance no heap allocations) is a requirement for many to accept the `variant`.
- Generality: a `variant` shall allow all types as alternatives.
- Regularity: a `variant` shall be as regular as its elements. Notably, it should be default constructible.
- Safety: a `variant` is a type-safe union. That should not come at the price of a built-in security hole in C++, with novice programmers triggering undefined behavior. The ideal `variant` is robust.

A `variant` can be invalid

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 proposed `variant` designs do have an additional invalid 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 have no object to access.

The type changing operation will need to construct an object in the internal buffer of the `variant`, likely using placement new, and pass the assigned object as an argument to the copy or move constructor. Thus, for the topic of invalid variants, only the behavior of copy and move constructors need to be taken into account.

How does `union` do it? C++ `union`s get around this by not managing type transitions. Assignments involving type transitions are too desirable to forbid them for `variant`. It's a crucial and fundamental feature.

Moving in Copy Assignment A `variant` can only become invalid in exceptional cases: during a type-changing assignment or emplacement, the copy or move constructor must throw. Generally, a copy constructor is more likely to throw than a move constructor. In the proposed designs, copying a type-changing alternative object into a variant will thus behave as if the object was copied into a temporary, and only upon success of that copy will the variant's old type be destructed and the temporary moved into the variant. This mechanism is only needed for type-changing assignments for which the copy constructor might throw. This further limits the cases for which invalidity needs to be handled.

In the subsequent discussion we can thus focus on throwing move constructors.

Banning assignment A `variant` could not offer assignment, neither of an alternative object (see for instance the `variant` presented at CPPCon 2014 (3)) nor of the variant object itself. Without, a `variant` obviously cannot undergo a type-changing assignment.

This is a huge restriction on the functionality.

Requiring `is_nothrow_move_constructible` This problem exists only for a subset of types. The exact set of types varies between implementations; implementations are free to add `noexcept`. Determining whether a type is `is_nothrow_move_constructible` is not trivial, especially for novice users, especially for types with multiple levels of inheritance and many data members, where all of these need to be checked. Change any member type to an `atomic`, or add a mutex, and the class cannot be stored in a variant anymore.

Even worse: a large fraction of C++98 "legacy" types have a copy but no move constructor. Putting any of those into a `pair`, `tuple`, etc renders the `pair`,

`tuple`, etc not `is_nothrow_move_constructible`. See Ville Voutilainen's paper "We cannot (realistically) get rid of throwing moves", expected in the pre-Kona mailing.

A `variant` is a really useful type. Limiting it to `is_nothrow_move_constructible` types reduces it to a few "good" alternative types; `variant` will become a powerful, simple tool for specialized use cases, instead of a powerful, mostly simple tool for general use.

Requiring the first alternative to be `is_nothrow_move_constructible` Peter Dimov suggested [http://lists.isocpp.org/lib-ext/2015/06/0132.php] that a `variant<T1,...>` that has undergone a throwing type-changing assignment will be set to a default-constructed `T1` alternative. This requires that `T1` is no throw default constructible, else assignment of an alternative that is not `is_nothrow_copy_constructible` will be ill-formed.

This conflates the invalid state with one of the valid states. Peter suggests to introduce an `empty` type as `T1` if the variant shall signal its error state.

This does not solve the problem for cases where the variant becomes invalid in one part of the program, and its value is tested in a completely different part of the program. Passing a `variant<int, vector<int>>` to a library, and not knowing whether the `int` can be trusted is outright dangerous; it's a poster child of a security issue. The argument that `empty` should have been inserted then is not convincing: on my platform, `vector<int>` is nothrow move constructible. Too bad it isn't on yours. To write platform-independent safe code, we would need to teach developers to always add `empty` as first alternative.

Changing `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 type 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 noexcept move constructible; users would need to specialize the (semi-) destructive move trait for their type. This is far too complex.

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

that the current value is now stored in the secondary buffer. The disadvantages of a secondary buffer are

- the address of the `variant`'s internal storage changes.
- 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; either way there is a possible runtime cost that is hard to predict for users, or an additional size of the `variant`, rendering it sub-optimal in several cases.

An implementation provided by Anthony Williams (4) tries to use non-throwing move and copy wherever possible, and only reverts to double buffering if at least two alternatives have no non-throwing move construction. This still leaves many `variant`s with double buffers.

We prefer other options that have a simpler performance pattern. (Up to) doubling the size of a `variant` in cases that hard difficult to predict for regular users would be a reason for many to continue to use `union` - after all it is exactly the size optimization for multiple alternatives that makes `variant` an interesting vocabulary type.

Valid but only partially specified In the proposals, the contained object in an invalid `variant` can be thought of as a moved-from object: it leaves the `variant` in a perfectly valid but only partially specified state.

This state needs to be visible: accessing its contents or visiting it will violate preconditions or throw (depending on the proposal); 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 might 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 just became invalid.

Instead, we prefer to make this state visible through the `index()` returning `tuple_not_found` and a usability feature `valid()`.

The "partially" in "partially specified" A variant that has undergone a throwing type-changing assignment will contain an object with unspecified state. But the `variant` knows about its state: `valid()` has to return `false`. Copying or moving such a variant has to be allowed (without undefined behavior), to enable scenarios where such a `variant` is part of a `vector`, and the `vector` is

resizing. Implementing this is not problematic, due to the visible state of the invalid `variant`.

Thus the an invalid `variant` contains an object of unspecified state while it itself has a well specified state (invalid).

No strong exception guarantee The variant cannot provide a strong exception guarantee if any of its alternatives can throw in move construction, unless double buffering is employed.

Empty state and default construction Default construction of a `variant` should be allowed, to increase usability for instance in containers. Options are: - default construct by default constructing the first alternative if that is default constructible - add a dedicated type as first alternative that explicitly enables default construction and adds an explicit new state for the default constructed `variant` - default construct into the invalid (or empty) state

A union default constructs the first member, similar to the first option here.

Alternatives: duplicate, missing, cv-qualified and references

`variant<int, int>` Multiple occurrences of identical types are allowed. They are distinct states; the `variant` can contain either the first or the second `int`. This models a discriminated union. For a `variant` with duplicate types in the alternatives, use of the interfaces that identify the alternative through a type template parameter (constructor, emplace, get, etc) is ill-formed. Instead, `get<0>` has to be used in place of `get<int>`; emplacement-with-index instead of assignment or type-based emplacement; and construction providing an emplacement hint (`emplaced_index<0>`) instead of a construction passing an `int`.

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

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

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

`variant<int&>` References are supported as alternative types. Assignment to such a value is ill-formed.

`constexpr` access

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

`noexcept` interfaces

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

Perfect Initialization

The proposals employ the same mechanisms for perfect initialization (5) as `optional`; see the discussion there. A constructor tag `emplaced_type` is used to signal the perfect forwarding constructor overload. For symmetry reasons, `emplace_index<I>` can be used to signal the initialization to the I-th alternative.

Heterogenous and Element Assignment, Conversion and Relational Operators

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

```
variant_with_element_less<float, int> vi(12), vf(14.);
assert(
  vi < 13. && 13. < vf && vf < vi &&
  R"quote(
    "Oh dear," says Arian, "I hadn't thought of that,"
    and promptly vanishes in a puff of logic.
)quote")
```

A possible later extension that seems to still keep `variant` as a regular type are comparison operators of the form and behavior

```
template <class T, class... Types>
bool operator==(const variant<Types...>& v, const T& t) {
    return v == variant<Types...>{t}; // exposition only
}
```

for all alternative types `T`. They can be implemented in a more performant way than what is suggested here. There have been voices questioning that this variant would still be regular, though no proof has been found to this date.

Assignment, Emplace

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

Access to 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.

Comparing The Two Proposals

Visibility of invalid state

The major difference of the two proposals is how much the invalid state is exposed. The `variant` "without undefined behavior" argues that this state exists and has to be dealt with; no rug is large enough to change that. It naturally plays the role of an additional state of the `variant` with N alternatives, turning a type with N states into one with N+1 states. It makes it even more visible by providing a `clear()` function.

The "rarely invalid" `variant` argues that this state should really never be reached. The `variant` will try to be helpful for this degenerate state, but not at the cost of cluttering user code by forcing an extra validity check to element accesses because the invalid state is too common. The latter is usually simply part of the check for the currently active alternative; the access pattern for both proposals is:

```
if (holds_alternative<int>(v))
  return get<int>(v);
```

Undefined behavior on value extraction != security

Even though the "rarely invalid" `variant` makes an invalid `variant` very unlikely, it can still exist. Developers cannot rely on a given `variant` to have nothrow move constructible alternatives: this might change over implementations and / or time. Now they have to options: given how rare an invalid `variant` is, they just ignore that state in their program. If it ever happens (for instance triggered by an evil person) then element accesses will trigger undefined behavior, which can cause invalid data read, which can in turn have security implications.

The alternative is to always handle the very unlikely invalid state. That feels as futile as checking the return value of `printf()` if it basically never happens; it will be hard to convince developers that it's nonetheless crucial. Suppose we would convince all developers to always check for validity: then we are back to the exposed state of the `variant` "without undefined behavior", at the additional cost of introducing undefined behavior in some conditions. And by the way, undefined behavior in one of the fundamental operations of one of the fundamental types of a language is clearly *not* desirable.

Default construction

If there is a visible invalid state then default constructing to that state is a simple and obvious solution. This is similar to `optional`, `ifstream`, `unique_ptr` - at least in spirit.

Default constructing to invalid makes sure that developers realize that a `variant` can be invalid.

Visitation

The "rarely invalid" `variant` will rely on the caller of the visitation to ensure that no invalid `variant` is passed; else the behavior is undefined.

For the `variant` "without undefined behavior", visitation will throw if an invalid `variant` is passed. The original proposal (N4218) suggested to call an visitor overload that takes no argument for an invalid `variant`. This is not compatible with multi-visitation and has thus been revised.

Composability

Focused types are good. It is desirable to separate the ideas of `optional` and `variant`; `optional<variant<int, float>>` is more descriptive than just `variant<int, float>`. This suggests the "rarely invalid" `variant` is the superior type.

Modeling the math

Type theory prefers the "rarely invalid" `variant` (and even more so the double-buffering one) because it does not add a new state. Developers coming from other languages will have a smoother transition to C++, at least it might be perceived as such.

Performance

The accesses to a `variant` need to check whether the currently active variant alternative corresponds to the alternative requested in the access, i.e. whether for `get<0>(v)`, the active alternative of `variant v` is the first one. An additional state introduces no relevant performance penalty. All interfaces exhibiting undefined behavior on the "rarely invalid" `variant` are designed to throw an exception already; here, too, the proposal "without undefined behavior" does not add a runtime cost.

Both designs need to capture throwing move construction of an alternative, not introducing any conceivable performance penalty for either side.

Conclusion

A `variant` has proven to be a useful tool. This paper shows the complexity behind it.

Acknowledgments

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

References

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

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

3. *Variant by Jason Lucas at CppCon* [online]. Available from: https://github.com/JasonL9000/cppcon14

4. *Variant by Anthony Williams* [online]. Available from: https://bitbucket.org/anthonyw/variant

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