Discriminating `unions`

The Long Road to `std::variant`

Andreas Weis

Munich C++ User Group, November 2015

What is this 'variant' people keep talking about?

- Aka. tagged union, disjoint union, discriminated union...
- Basically a strongly-typed `union`. Think:

```
enum class Tag { Integer, Float, String };
struct Variant {
  Tag tag;
  union Value {
    int i;
    float f;
    std::string str;
  } value;
};
```

- Algebraic sum data type. Dual to `std::tuple`.

Example

```
variant<int, float, std::string> v = 42;

int n = get<int>(v);

v = std::string("Narf");
std::string s = get<string>(v);

float f = get<float>(v);  // *boom*
```

Example

```
struct Circle { float getArea() const; };
struct Square { float getArea() const; };
struct Triangle { float getArea() const; };
typedef variant<Circle, Square, Triangle> Shape;

struct GetShapeAreaVisitor {
  template<typename T>
  float operator()(T const& s) {
    return s.getArea();
  }
};

float getArea(Shape const& s) {
  return visit(GetShapeAreaVisitor{}, s);
}
```

Previous Work

- Alexandrescu
  - *An Implementation of Discriminated Unions in C++*, Template Programming Workshop, OOPSLA 2001
  - Dr. Dobb's Generic Programming: Discriminated Unions, 2002
- Friedman - Boost.Variant, 2003
- Berg - Eggs.Variant, 2014

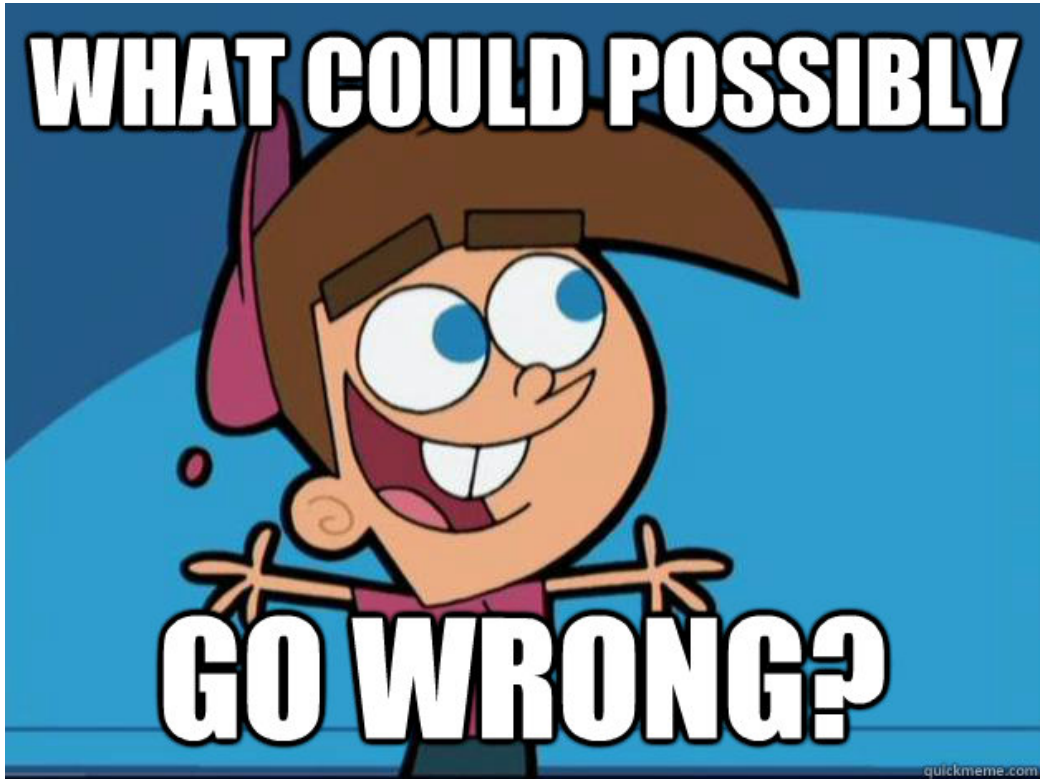
Early approaches were constrained by C++98.

People love variants!

Someone should write a proposal...

People love variants!

Someone should write a proposal...



Refresher: Exception Safety Guarantees

Possible exception safety guarantees for a function `func()`:

Refresher: Exception Safety Guarantees

Possible exception safety guarantees for a function `func()`:

- `noexcept` - `func()` will not throw an exception.

Refresher: Exception Safety Guarantees

Possible exception safety guarantees for a function `func()`:

- `noexcept` - `func()` will not throw an exception.
- *Strong exception guarantee* - If `func()` throws, the program state will be left unchanged (as if the call never happened).

Refresher: Exception Safety Guarantees

Possible exception safety guarantees for a function `func()`:

- `noexcept` - `func()` will not throw an exception.
- *Strong exception guarantee* - If `func()` throws, the program state will be left unchanged (as if the call never happened).
- *Basic exception guarantee* - If `func()` throws, the program will be left in a valid state (for member functions, usually allows at least assignment-to and destruction of the object).

Variant assignment

```
std::string teststring("Lorem ipsum dolor sit");

std::variant<int, std::string> v1(42);
v1 = teststring;
```

What if the copy assignment for `std::string` throws?

This is different from `std::optional`! We switch types during assignment.


CHALLENGE ACCEPTED

What would Boost do?

- `boost::variant` gives a *never-empty* guarantee.
- For a `variant<T1,T2,...,TN>` one of the `T`s is always held in storage.
- Default construction always default-constructs `T1`.
- Uses *double-buffering* for assignment.
- A temporary storage is allocated on the heap and holds a copy of the original data while the assignment takes place.

Disadvantages: Heap allocation required. Order of operations is unintuitive.

N4218 - Variant v1

First proposal by Axel Naumann in September 2014.

- Double-buffering was deemed too inefficient.
- `variant` reverts to an empty state upon default construction and throwing assignment.
- Emptyness can be queried through `empty()` member function.
- `get()` will throw when empty.

Disadvantages?

N4450 - Variant v2

Naumann, April 2015.
In Urbana it was decided that the initial variant TS should include visitors.
How to visit a variant that can be empty?

```
struct Visitor {
  template<typename Ti>
  void operator()(Ti) { /* variant types */ }

  void operator()() { /* empty type */ }
};

variant v;
visit(Visitor(), v);
```

N4450 - Variant v2

```
struct Visitor {
  template<typename Ti>
  void operator()(Ti) { /* variant types */ }
};

struct EmptyVisitor {
  void operator()() { /* empty type */ }
};

variant v;
visit(Visitor(), EmptyVisitor(), v);
```

N4516, N4242 - Variant v3/v4

Naumann, May 2015.
Variants were heavily discussed in Lenaxa, which led to a number of changes.

- `!empty()` becomes `valid()`.
- The invalid state is restricted to failed assignments.
- Default construction now uses the first type (just like `union` and `boost::variant`). Type `monostate` to explicitly enable default construction.
- Accessing the invalid state is UB. The empty visitor goes away.

The change from empty to invalid is bigger than it might seem at first glance.

A. Williams - Standardizing Variant: Difficult Decisions

Blog post on Williams' private blog in June 2015. Spawns numerous discussions in the blog comments, on Reddit, and the ISO mailing list.

- Williams argues that UB for accessing invalid variants is too hard to check.
- `get()` should throw instead. Copying from an invalid variant should be possible
- Niebler argues that invalid is not a problem as it can only happen if an assignment fails with exception.

Surely they will be able to figure this out before Kona...

p0088r0 - Variant v5 - a type-safe union that is rarely invalid

- Mostly a cleaned-up version of v4.
- Completely removes the notion of the empty type.
- Invalid state is observable through member functions `valid()` and `index()`.
- Accessing the invalid state is still UB.
- But invalid variants can now be assigned-from.

p0080r - Discriminated Union with Value Semantics

July 2015, Alternative proposal by Michael Park

- After throwing assignment *everything* except assign-to and destruct is UB. Unlike N4542, indeterminate state here is *unobservable*.
- Default construct to indeterminate state.
- New `nullstate` type allows to opt-in for a safe fallback type.
- Visitation is handled completely different.

p0093r0 - Simply a Strong Variant

David Sankel, September 2015

- Argues that invalid state is problematic as it has no analogy in mathematical discriminated unions.
- Strong exception guarantee is easy if types allow non-throw move construction.
- Use in-place double-buffering if one of the alternative types can throw on move-construction.
- Argues that the user can always declare `noexcept` move constructor to get rid of double-buffering.

Basically an old-school `boost::variant` with some C++11 tuning.

p0094r0 - Simply a Basic Variant

David Sankel, September 2015

- Who needs strong exception safety anyway?
- If assignment throws, default construct one of the alternative types.
- Never empty is guaranteed, though we don't know what will be in there.
- Use double-buffering as fallback if none of the types is nothrow default constructible.

...and more!

- p0095r0 - Sankel, *The Case for a Language Based Variant*: We are doomed! Compiler writers, save us!
- p0087r0 - Naumann, *A type-safe union without undefined behavior*: Variant v2-b. Maybe we need to go back to where we started.
- p0110 - Williams, *Implementing the strong guarantee for variant assignment*: Do the most efficient thing possible given the nothrow guarantees of the types. Still has to fallback to double-buffering in pathological case.

Total of 8 proposals discussing variant in September 2015.

The Kona Kompromise - A happy ending?

Kona Trip Report by Naumann, October 2015.[1]

- Basically v5, but `get()` will throw if invalid.
- No more undefined behavior!
- No more `assert(v.valid())` cluttering your code!
- Strong exception safety for non-throwing move; Basic exception safety otherwise.
- "Almost everyone in the room was happy!"

Re-review pending for the Jacksonville meeting in March 2016.

[1] `https://isocpp.org/blog/2015/11/the-variant-saga-a-happy-ending`

Conclusion

- There is no obvious right answer for default construction and throwing assignment.
- People do care about variant.
- Discussions can be chaotic, but will be worth it if we can work it out.

Thanks for your attention.