The Many Variants of `std::variant`

Nevin ":-)" Liber
nliber@anl.gov

Argonne NATIONAL LABORATORY

This presentation was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative. Additionally, this presentation used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

What is a variant?

- Discriminated/tagged/typesafe union
  - Union that knows what type it is holding
- Type theory algebraic types
  - Product type (n-ary)
    - `struct`/`class`
    - Generic: `tuple` (C++11)
  - Sum type (1 of n)
    - `union`
    - Generic: `variant` (C++17)

Three vocabulary sum types

- any
- variant
- optional

Three vocabulary sum types

- **any**
- variant
- optional

any

- Concrete type
- Discriminated type
- Open sum type
  - The set of acceptable types is unbounded
- Type requirements

any

- Cpp17CopyConstructible requirement
  - (Cpp17 requirements are syntactic, not semantic requirements)
- Type erasure
  - Generates (the equivalent of) a `virtual` function for most operations
    - `virtual` function for cloning the object
    - Even if the object is never copied

```
class any {
    struct Concept {
        virtual ~Concept() = default;
        virtual Concept* clone() const = 0;
        //...
    };

    template<typename T>
    struct Model final : Concept {
        template<typename... Args>
        Model(Args&&... args) : data(std::forward<Args>(args)...) {}

        Model* clone() const override { return new Model(data); }
        //...
        std::decay_t<T> data;
    };
    std::unique_ptr<Concept> object;    // holds a Model<T> pointer
public:
    any() noexcept = default;
    any(any const& that) : object(that.object ? that.object->clone() : nullptr) {}
    any(any&&) noexcept = default;
    //...
    template<typename T, typename... Args>
    std::decay_t<T>& emplace(Args&&... args) {
        object.reset();
        std::unique_ptr<Model<T>> modelT =
            std::make_unique<Model<T>>(std::forward<Args>(args)...);
        std::decay_t<T>& t = modelT->data;
        object = std::move(modelT);
        return t;
    }
};
```

```
class any {
    struct Concept {
        virtual ~Concept() = default;
        virtual Concept* clone() const = 0;
        //...
    };

    template<typename T>
    struct Model final : Concept {
        template<typename... Args>
        Model(Args&&... args) : data(std::forward<Args>(args)...) {}

        Model* clone() const override { return new Model(data); }
        //...
        std::decay_t<T> data;
    };

    std::unique_ptr<Concept> object;    // holds a Model<T> pointer
public:
    any() noexcept = default;
    any(any const& that) : object(that.object ? that.object->clone() : nullptr) {}
    any(any&&) noexcept = default;
    //...
    template<typename T, typename... Args>
    std::decay_t<T>& emplace(Args&&... args) {
        object.reset();
        std::unique_ptr<Model<T>> modelT =
            std::make_unique<Model<T>>(std::forward<Args>(args)...);
        std::decay_t<T>& t = modelT->data;
        object = std::move(modelT);
        return t;
    }
};
```

- `make_unique` instantiates a `Model<T>`
  - Code generated for every `virtual` function in `Model<T>`
    - clone()
      - Calls copy constructor for *`decay`*`<T>` on data
      - Still generated even if `any(any const&)` didn't exist

```
template<typename T>
struct Model final : Concept {
    template<typename... Args>
    Model(Args&&... args) : data(std::forward<Args>(args)...) {}

    Model* clone() const override { return new Model(data); }
    //...
    std::decay_t<T> data;
};
```

```
    std::unique_ptr<Concept> object;    // holds a Model<T> pointer
public:
    any() noexcept = default;
    any(any const& that) : object(that.object ? that.object->clone() : nullptr) {}
    any(any&&) noexcept = default;
    //...
    template<typename T, typename... Args>
    std::decay_t<T>& emplace(Args&&... args) {
        object.reset();
        std::unique_ptr<Model<T>> modelT =
            std::make_unique<Model<T>>(std::forward<Args>(args)...);
        std::decay_t<T>& t = modelT->data;
        object = std::move(modelT);
        return t;
    }
};
```

```
class any {
    struct Concept {
        virtual ~Concept() = default;
        virtual Concept* clone() const = 0;
        //...
    };

    template<typename T>
    struct Model final : Concept {
        template<typename... Args>
        Model(Args&&... args) : data(std::forward<Args>(args)...) {}

        Model* clone() const override { return new Model(data); }
        //...
        std::decay_t<T> data;
    };
    std::unique_ptr<Concept> object;    // holds a Model<T> pointer
public:
    any() noexcept = default;
    any(any const& that) : object(that.object ? that.object->clone() : nullptr) {}
    any(any&&) noexcept = default;
    //...
    template<typename T, typename... Args>
    std::decay_t<T>& emplace(Args&&... args) {
        object.reset();
        std::unique_ptr<Model<T>> modelT =
            std::make_unique<Model<T>>(std::forward<Args>(args)...);
        std::decay_t<T>& t = modelT->data;
        object = std::move(modelT);
        return t;
    }
};
```

```
class any {
    struct Concept {
        virtual ~Concept() = default;
        virtual Concept* clone() const = 0;
        //...
    };

    template<typename T>
    struct Model final : Concept {
        template<typename... Args>
        Model(Args&&... args) : data(std::forward<Args>(args)...) {}

        Model* clone() const override { return new Model(data); }
        //...
        std::decay_t<T> data;
    };

    std::unique_ptr<Concept> object;    // holds a Model<T> pointer
public:
    any() noexcept = default;
    any(any const& that) : object(that.object ? that.object->clone() : nullptr) {}
    any(any&&) noexcept = default;
    //...
    template<typename T, typename... Args>
    std::decay_t<T>& emplace(Args&&... args) {
        object.reset();
        std::unique_ptr<Model<T>> modelT =
            std::make_unique<Model<T>>(std::forward<Args>(args)...);
        std::decay_t<T>& t = modelT->data;
        object = std::move(modelT);
        return t;
    }
};
```

- Space and runtime efficiency concerns
  - Held object may be allocated on the heap
    - Object too big for small object optimization
      - Embedded space inside any
    - If the move constructor for `T` can `throw`
      - Because `any(any&&)` is `noexcept`
        - Will dive into why this is desirable
    - No control over allocation
      - Not known how to store a type erased allocator in a type erased class

```
    std::unique_ptr<Concept> object;    // holds a Model<T> pointer
public:
    any() noexcept = default;
    any(any const& that) : object(that.object ? that.object->clone() : nullptr) {}
    any(any&&) noexcept = default;
    //...
    template<typename T, typename... Args>
    std::decay_t<T>& emplace(Args&&... args) {
        object.reset();
        std::unique_ptr<Model<T>> modelT =
            std::make_unique<Model<T>>(std::forward<Args>(args)...);
        std::decay_t<T>& t = modelT->data;
        object = std::move(modelT);
        return t;
    }
};
```

```
class any {
    struct Concept {
        virtual ~Concept() = default;
        virtual Concept* clone() const = 0;
        //...
    };

    template<typename T>
    struct Model final : Concept {
        template<typename... Args>
        Model(Args&&... args) : data(std::forward<Args>(args)...) {}

        Model* clone() const override { return new Model(data); }
        //...
        std::decay_t<T> data;
    };
    std::unique_ptr<Concept> object;    // holds a Model<T> pointer
public:
    any() noexcept = default;
    any(any const& that) : object(that.object ? that.object->clone() : nullptr) {}
    any(any&&) noexcept = default;
    //...
    template<typename T, typename... Args>
    std::decay_t<T>& emplace(Args&&... args) {
        object.reset();
        std::unique_ptr<Model<T>> modelT =
            std::make_unique<Model<T>>(std::forward<Args>(args)...);
        std::decay_t<T>& t = modelT->data;
        object = std::move(modelT);
        return t;
    }
};
```

any

- Runtime efficiency concerns
  - O(n) to find the type it holds
  - Cascading `if` statements

```
std::any a = AnyFactory();
if (int* i = std::any_cast<int>(&a))
    ProcessInt(*i);
else if (std::string* s = std::any_cast<std::string>(&a))
    ProcessString(*s);
// else if ...
```

Why is noexcept move construction desirable?

- Exception safety guarantee (Dave Abrahams)

  - Basic - invariants preserved, no resources leaked

  - Strong - operation either completed successfully or threw an exception. If the latter, the program state hasn't been changed. May be expensive or impossible to achieve

  - No-throw - operation will not throw an exception

Why is noexcept move construction desirable?

- Optimization when we need the strong guarantee
  - E.g.: Growing the internal size of a vector
    - Allocate new space
    - Throwing move
      - Copy elements
        - If exception, destroy all elements in new space and release it
        - Otherwise, destroy all elements in old space and release old space
      - Non-throwing move
        - Move elements and release old space

Why is noexcept move construction desirable?

- Optimization when we need the nothrow guarantee
  - E.g. swap:

```
template<typename T>
constexpr void swap(T&l, T& r)
noexcept(is_nothrow_move_constructible_v<T> && is_nothrow_move_assignable_v<T>)
{
    T t(std::move(l));
    l = std::move(r);
    r = std::move(t);
}
```

  - If T is noexcept move constructible/assignable, swap is automatically also noexcept

Why is noexcept move construction desirable?

- Throwing move constructors don’t play nicely with `variant`
  - Much, much more on that…

Three vocabulary sum types

- any
- **variant**
- optional

variant

- Templated type
- Discriminated type
- Closed sum type
  - Holds 1 of n possible types

variant

- The set of types is specified
  - Space efficiency
    - We know the amount of space needed
    - Space embedded in variant
- Runtime efficiency
  - O(1) access to the object it holds
    - Visitation

Three vocabulary sum types

- any
- variant
- **optional**

optional

- Templated type
  - Closed sum type
    - Holds at most one of the templated type
    - Refinement of `variant`
      - Easier interface
        - Eg: `*o` (dereference) to access object

C++ Committee

C++ Committee

C++ Committee

- Every committee member wants to make C++ better
  - Even if no two of us can agree on what that is
- Consensus of countries
  - One country, one vote
  - “You can’t always get what you want, but if you try sometime, you find, you get what you need.” - *The Rolling Stones*
  - Fight for what you can live with
    - Not necessarily your ideal design

C++ Committee

- The details matter
  - Millions of developers will use these classes
    - Millions of developers will model their types on the way the committee designs its types
    - It is very hard to fix things later
      - High bar for backwards compatibility

Designing variant

- Never-empty
- Assignment
- Comparisons
- Default construction
- Accessing its element

Tradeoffs

- Space efficiency
- Runtime efficiency
- Compile time efficiency
- Usability

Never-empty

- Ideally, variant has the never-empty guarantee
  - Models ***exactly-1*** of the bound types
- What if we relax it to modeling ***at-most-1***?
  - How does it affect the various tradeoffs?

Assignment

```
variant<A, B> v1 = A();
v1 = A();
```

- If `A` has an assignment operator, should we use it?
  - Assumes assignment is an optimization over destroy-then-construct
- If `A` doesn't have an assignment operator, should we destroy-then-construct?
- Should we always destroy-then-construct?
  - Consistency

Assignment

```
variant<A, B> v1 = A();
v1 = B();
```

- emplace has similar issues
- User expects resultant `v1` to hold a `B` object
  - Destroy-then-construct
  - What happens if `B()` throws?
    - ***Most contentious question!***

Comparisons

- Compare values

```
variant<short, int> vs3 = static_cast<short>(3);
variant<short, int> vi2 = static_cast<int>(2);
assert(vi2 < vs3);
```

- Compare alternative-then-values

```
variant<short, int> vs3 = static_cast<short>(3);
variant<short, int> vi2 = static_cast<int>(2);
assert(vs3 < vi2);
```

Comparisons

- Compare values

```
variant<short, int> vs3 = static_cast<short>(3);
variant<short, int> vi2 = static_cast<int>(2);
assert(vi2 < vs3);
```

- Result not surprising
    - 2 < 3

Comparisons

- Compare values

```
variant<short, int> vs3 = static_cast<short>(3);
variant<short, int> vi2 = static_cast<int>(2);
assert(vi2 < vs3);
```

- Need entire matrix of comparisons (O(n^2))
  - What happens with variant<int, string>?
    - Not compile comparisons (SFINAE or hard error)
    - Throw exception when comparing int with string

Comparisons

- Compare values of different types can break transitivity

```
// Lexicographically compare ints and strings
bool operator<(int i, string s){ return to_string(i) < s; }
bool operator<(string s, int i){ return s < to_string(i); }
//…

using V = variant<int, string>;
V vi = 200;
V vs = "30";
V vj = 5;
assert(vi < vs);    // 200 < “30” because “200” < “30”
assert(vs < vj);    // “30” < 5 because “30” < “5"
assert(vi < vj);    // fails because 200 > 5
```

- This *is* surprising!

Comparisons

- Compare values

```
variant<int, int> v0(in_place_index<0>, 5);
variant<int, int> v1(in_place_index<1>, 5);
assert(v0 == v1);
```

- This *is* surprising
  - Are they really equal?
  - Alternative is not a salient feature of value comparing variant

Comparisons

- Compare alternative-then-values
  - “Representational” comparison

```
variant<short, int> vs3 = static_cast<short>(3);
variant<short, int> vi2 = static_cast<int>(2);
assert(vs3 < vi2);
```

- Result *is* surprising
- Types only need be self-comparable (O(n))

Default Construction

- Should it be default constructible?
- If it is default constructible, should it default construct one of its bound types?
- Is this the same as the moved-from state?

Accessing its element

- Index by type?
- Index by alternative?
- Should we have a better than O(n) runtime (ideally O(1)) way to access the element?
  - Visitation

Let's Brainstorm!

```
variant<A, B> v1 = A();
v1 = B();
```

- `v1` holds an object of type `A`
- We want to store an object of type `B`
- Constructor for `B` throws

Boost.Variant

- Added to Boost in 1.31 (2004)
- Never-empty guarantee
- Assignment is… interesting
- Default constructible… sometimes
- Compares alternative-then-values
- Visitation

Boost.Variant Assignment

```
boost::variant<A, B> v1 = A();
v1 = A();
```

- Uses `A::operator=`

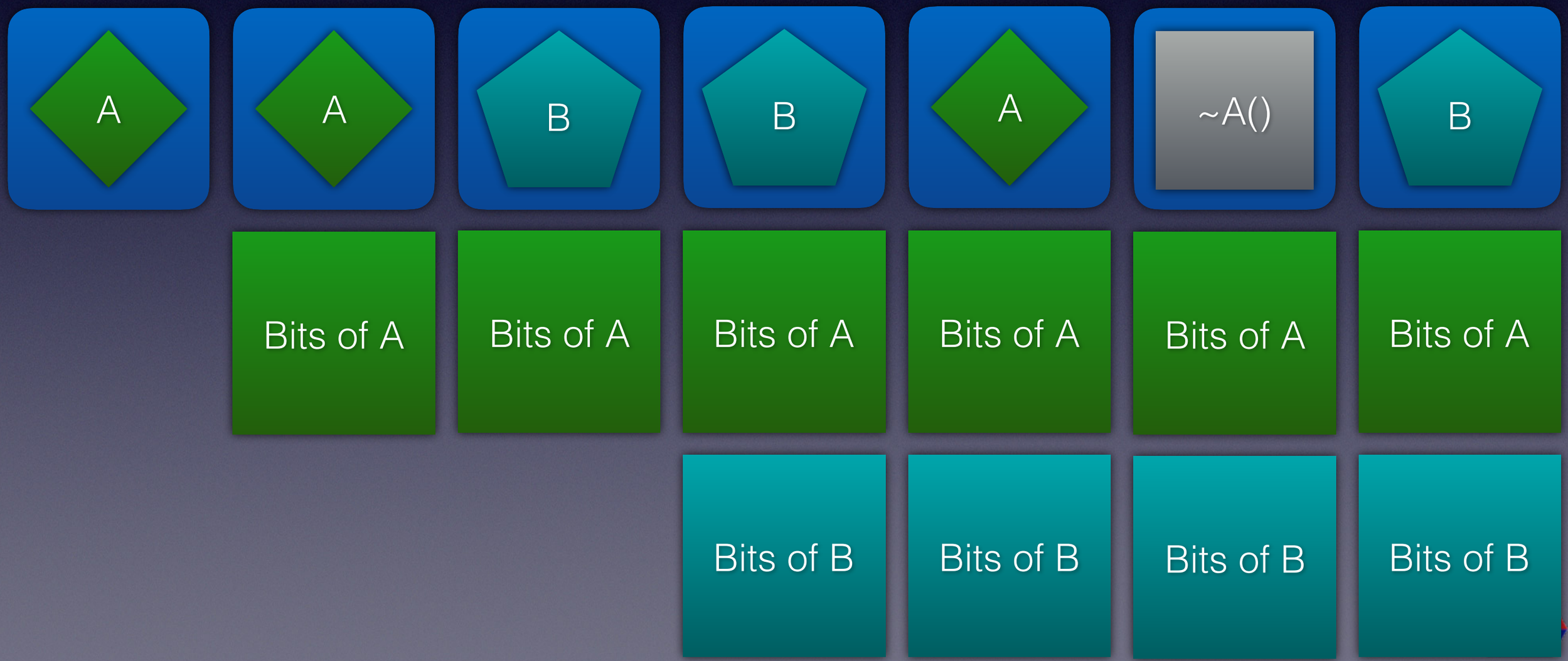
“Ideal” Solution: False Hopes

```
boost::variant<A, B> v1 = A();
v1 = B();
```

1. Provide “backup” and “temporary” storage
2. memcpy() v1 to backup storage
3. Attempt to move construct B into v1
    3.1. If exception, memcpy() backup into v1 and done
4. memcpy() v1 to other shared “temporary” storage
5. memcpy() backup into v1. v1 now contains A
6. Destroy A in v1
7. memcpy() temporary into v1. V1 now contains B

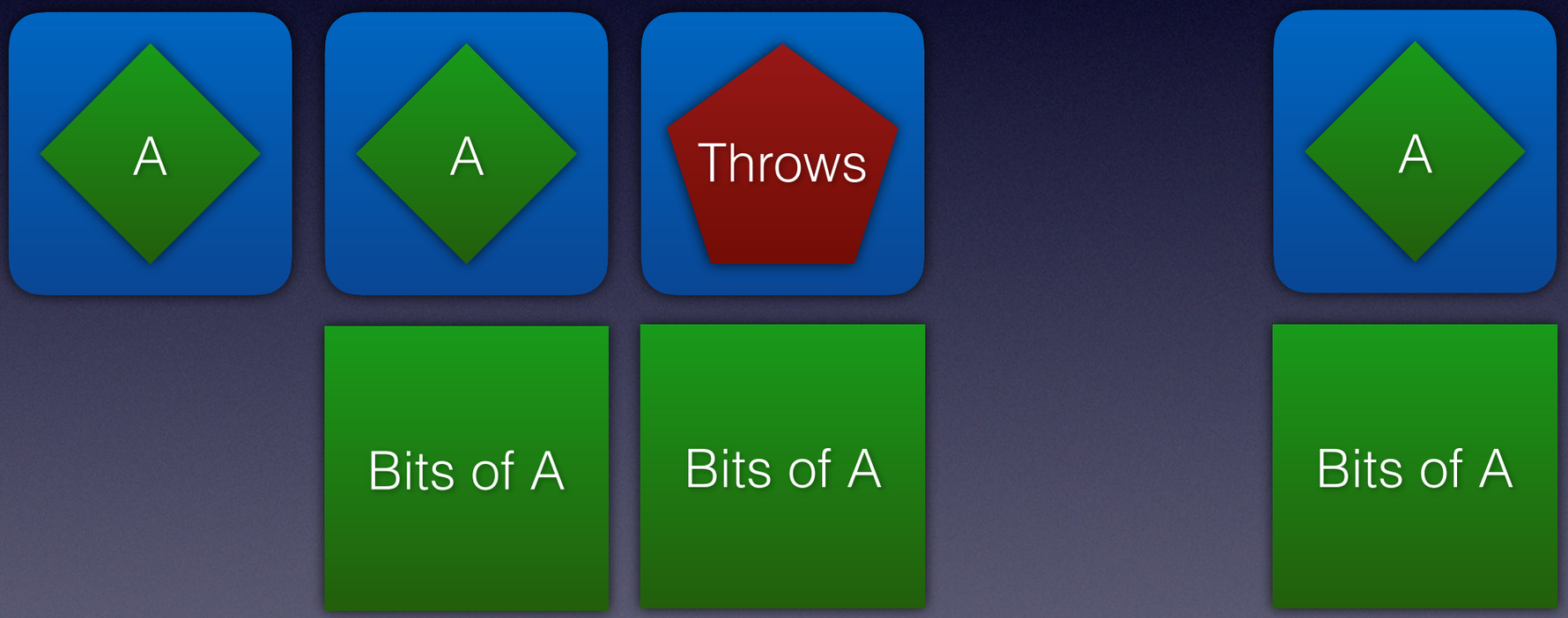
“Ideal” Solution: False Hopes

```
boost::variant<A, B> v1 = A();
v1 = B();
```



“Ideal” Solution: False Hopes

```
boost::variant<A, B> v1 = A();
v1 = B();
```



“Ideal” Solution: False Hopes

```
boost::variant<A, B> v1 = A();
v1 = B();
```

- Fraught with peril!
- “That's a lot of code to read through, but if it's doing what I think it's doing, it's ***undefined behavior***.
- "Is the trick to move the bits for an existing object into a buffer so we can tentatively construct a new object in that memory, and later move the old bits back temporarily to destroy the old object?
- "The standard does not give license to do that: only one may have a given address at a time…” - Dave Abrahams

Initial Solution: Double Storage

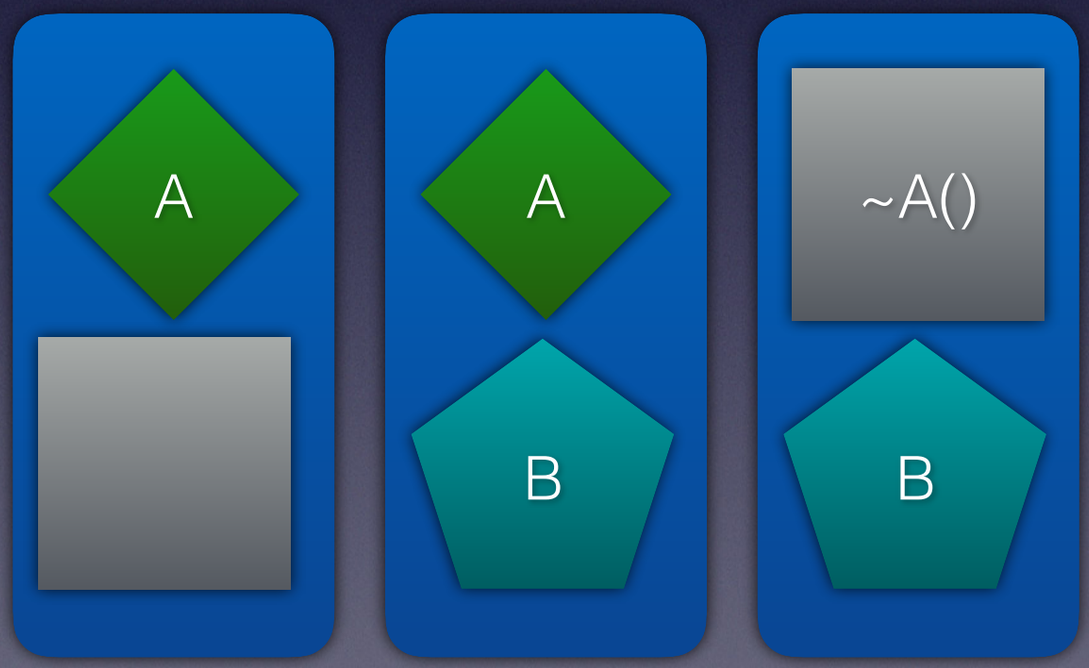
```
boost::variant<A, B> v1 = A();
v1 = B();
```

- Construct-then-destroy
- ***Completely solves the problem***
- Strong exception safety guarantee
- get<A>(v1) may give a different address depending on when it is called
- But…
  - ***Space overhead unacceptable for many users***

Initial Solution: Double Storage

```
boost::variant<A, B> v1 = A();
v1 = B();
```

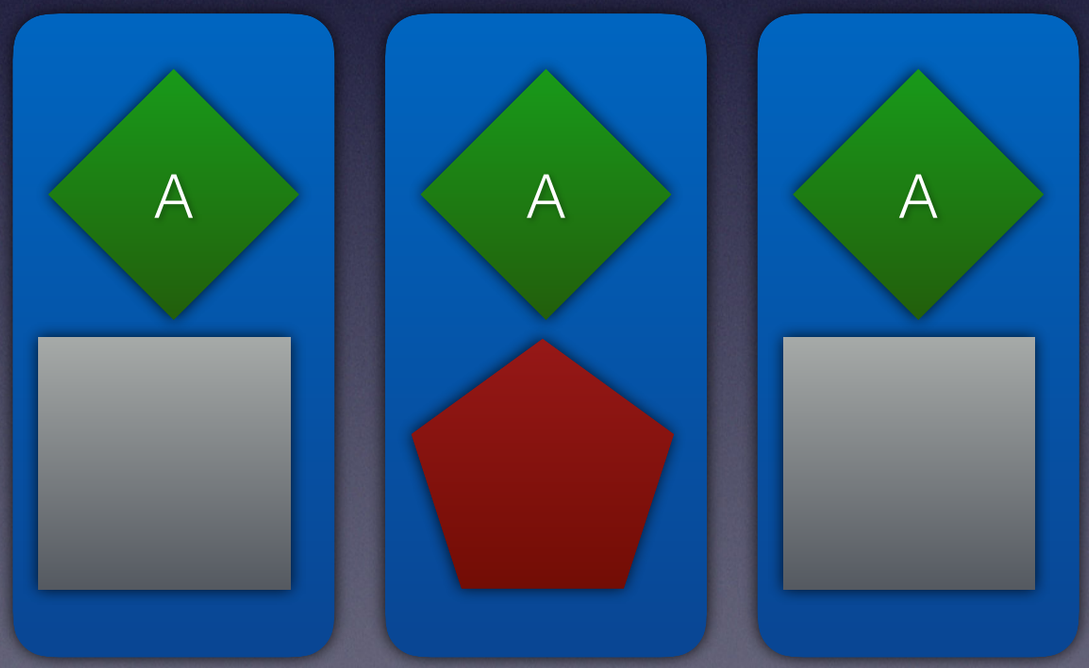
- Construct-then-destroy



Initial Solution: Double Storage

```
boost::variant<A, B> v1 = A();
v1 = B();
```

- Construct-then-destroy



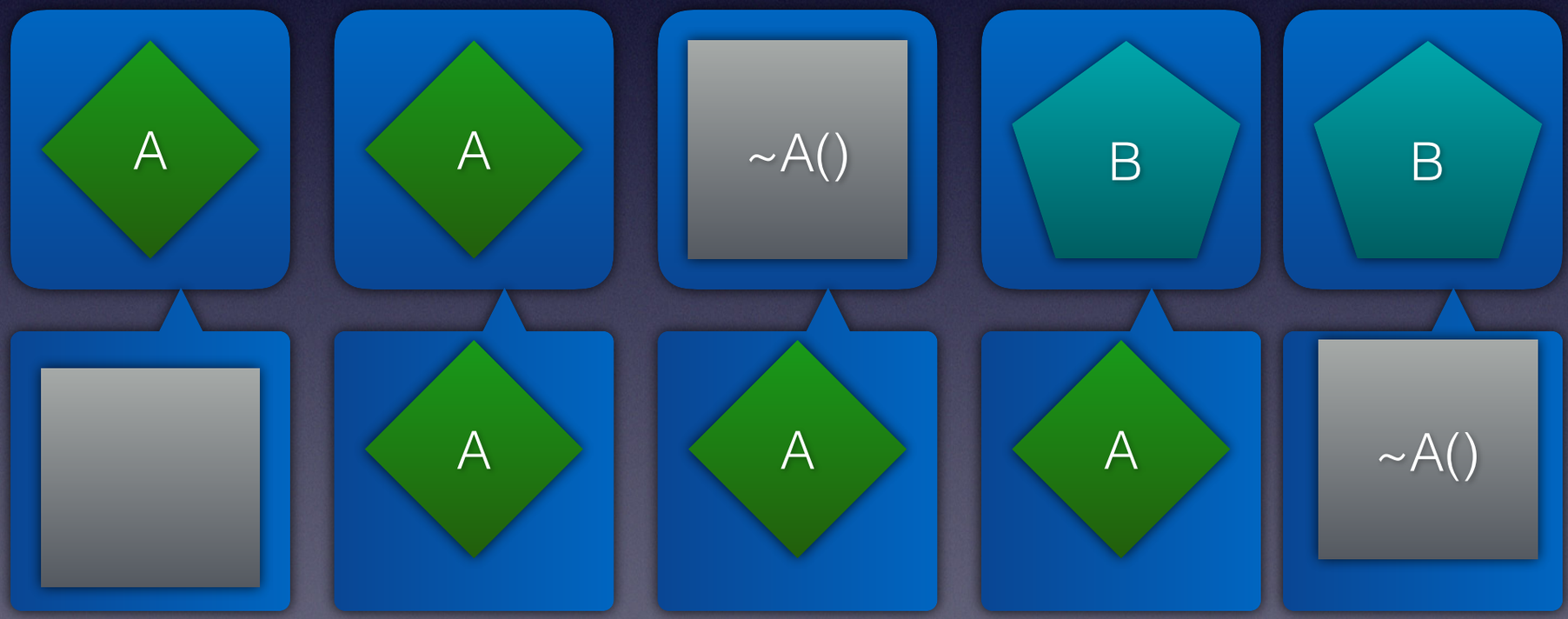
Current Approach: Temporary Heap Backup

```
boost::variant<A, B> v1 = A();
v1 = B();
```

1. Destroy-then-construct
2. If any type is noexcept default constructible, only single storage is used
3. Otherwise, heap backup
    1. Copy construct v1 to heap backup
    2. Destroy v1
    3. Construct B into v1
        1. If exception, use backup
        2. Otherwise, destroy A in backup

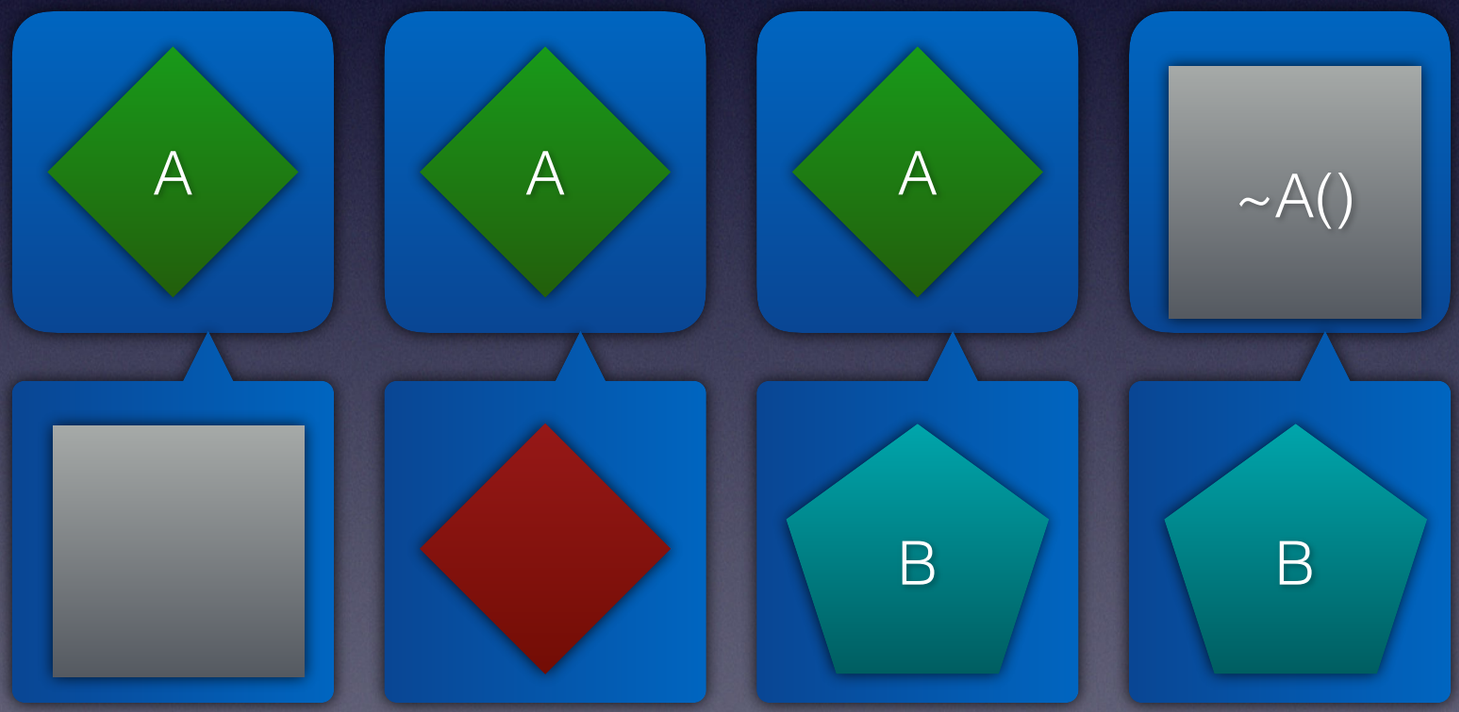
Temporary Heap Backup

```
boost::variant<A, B> v1 = A();
v1 = B();
```



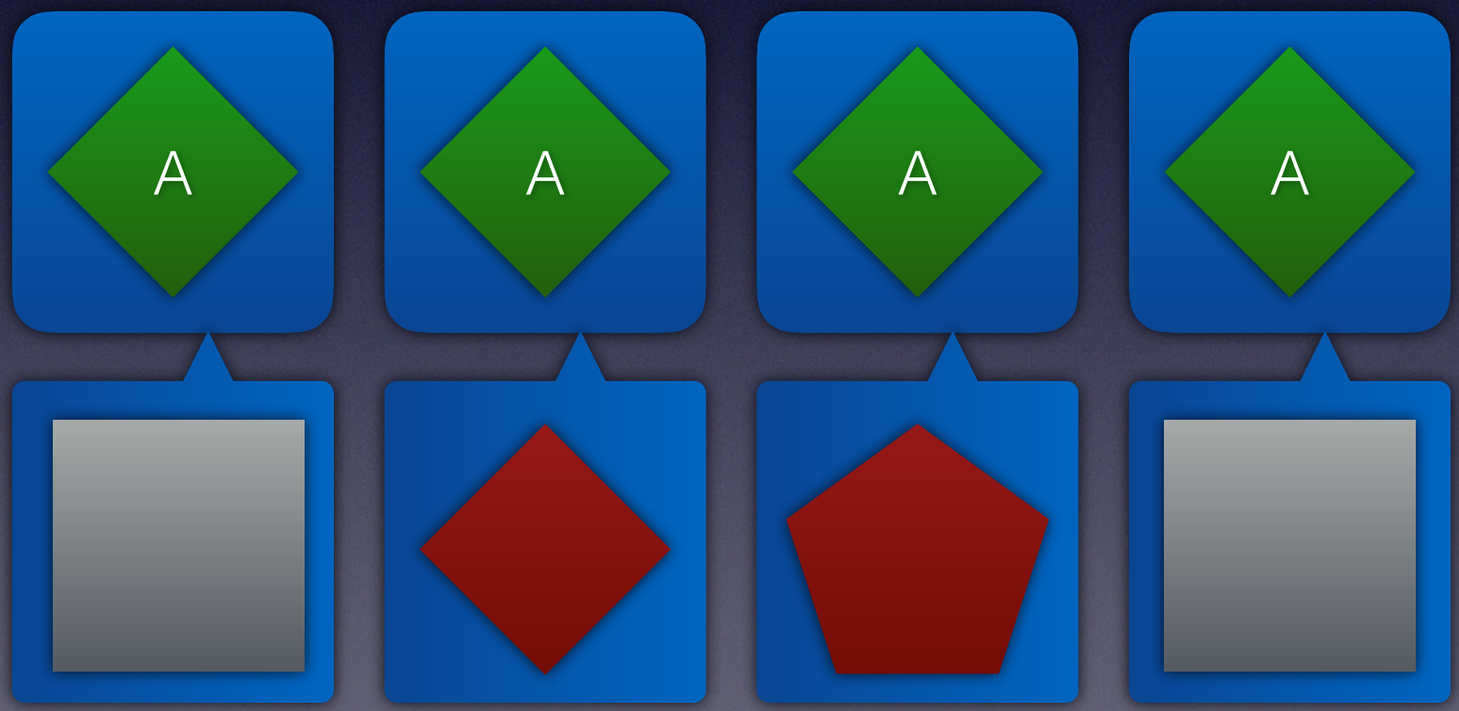
Temporary Heap Backup

```
boost::variant<A, B> v1 = A();
v1 = B();
```



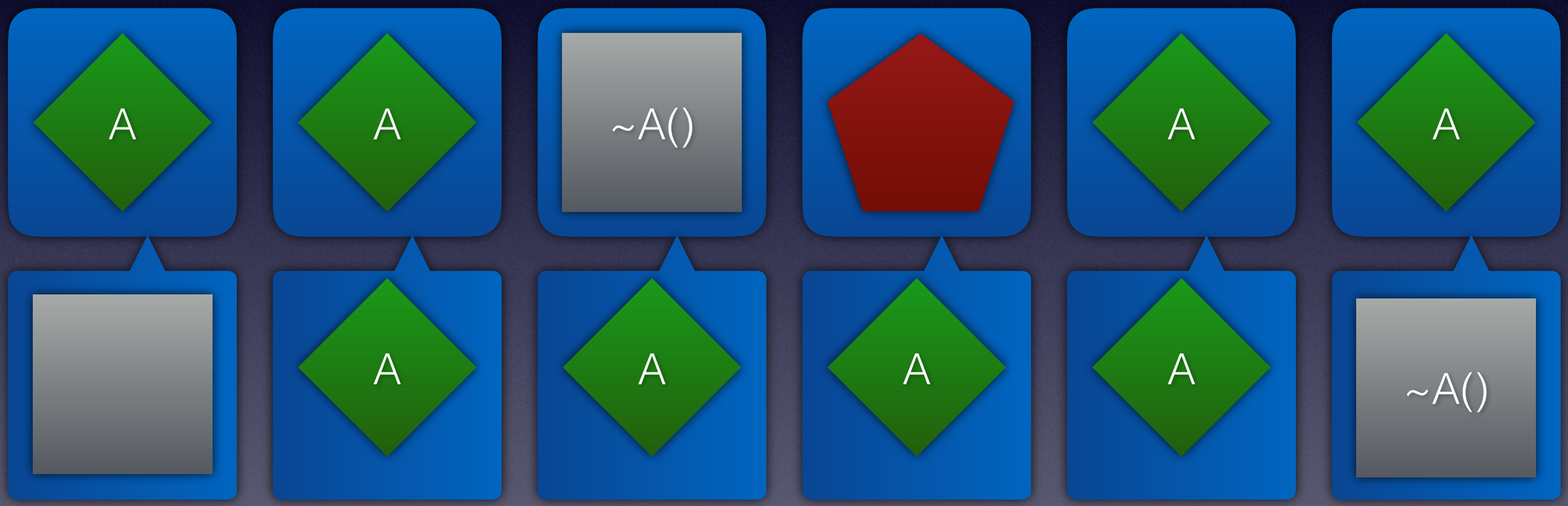
Temporary Heap Backup

```
boost::variant<A, B> v1 = A();
v1 = B();
```



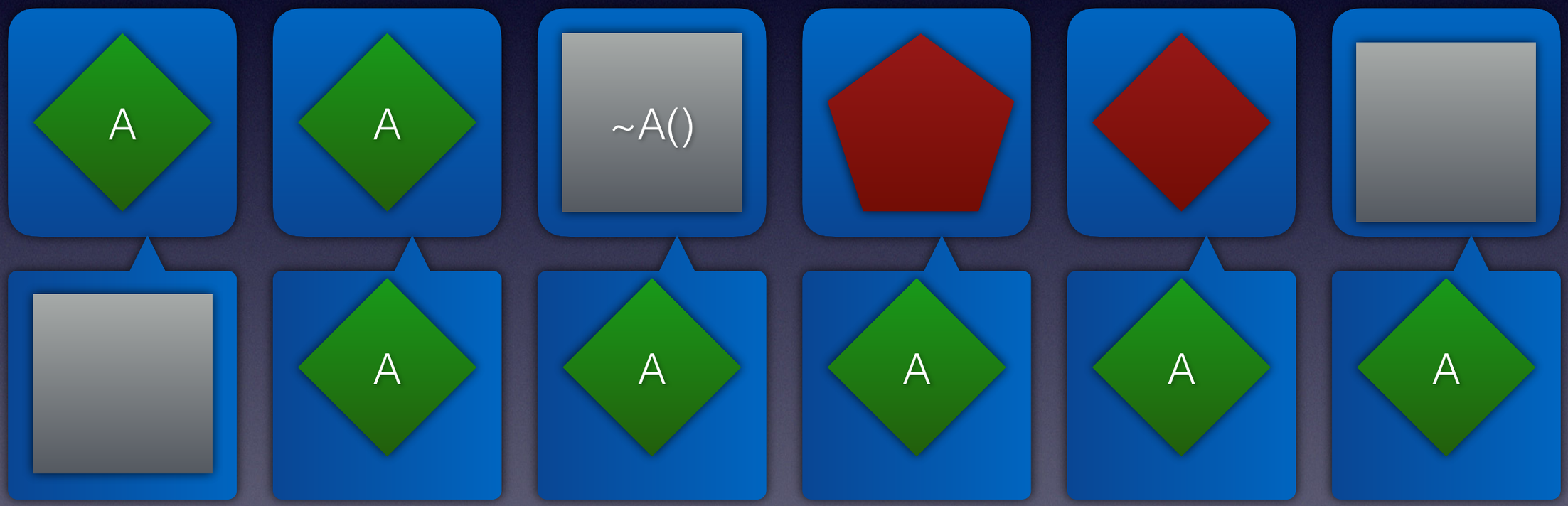
Temporary Heap Backup

```
boost::variant<A, B> v1 = A();
v1 = B();
```



Temporary Heap Backup

```
boost::variant<A, B> v1 = A();
v1 = B();
```



Default Constructible

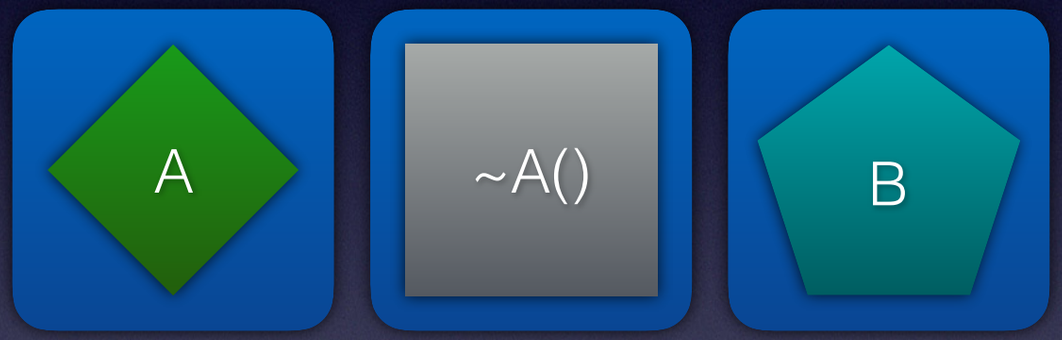
- If the first type is default constructible, then boost::variant is default constructed with that type

- If any type is noexcept default constructible and an exception is thrown when changing the held type, the variant is left holding an unspecified noexcept default constructible type

  - boost::blank is preferred

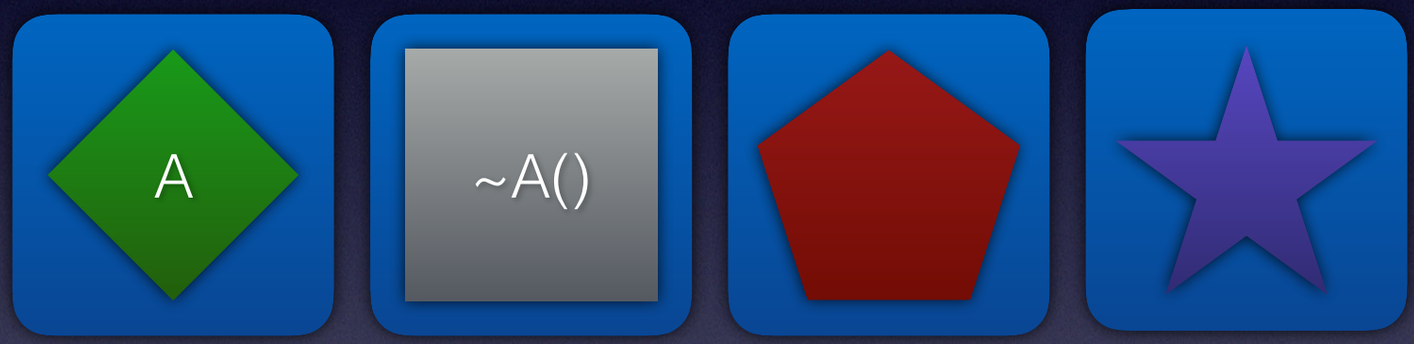
Nothrow Default Constructible

```
boost::variant<A, B, boost::blank> v1 = A();
v1 = B();
```



Nothrow Default Constructible

```
boost::variant<A, B, boost::blank> v1 = A();
v1 = B();
```



Element access

- get
  - Index by type
  - Similar to any_cast for any

```
void printViaGet(variant<std::string, int>& v) {
    if (std::string* s = get<std::string>(&v))
        std::cout << "string: " << *s << '\n';
    else if (int* i = get<int>(&v))
        std::cout << "int: " << *i << '\n';
}
```

Element access

- Visitation
  - Function object which knows how to process every type
  - apply_visitor(vis, var)
    - Passes held object to corresponding function call operator
  - Can be O(1) over the number of types

Element access

- Visitation

```
struct printVis : boost::static_visitor<> {
    void operator()(std::string s) const
    { std::cout << "string: " << s << '\n'; }
    void operator()(int i) const
    { std::cout << "int: " << i << '\n'; }
};
void printViaVis(V& v)
{ apply_visitor(printVis(), v); }
```

- Inversion of control

Element access

- Get

```
void printViaGet(variant<std::string, int>& v) {
    if (std::string* s = get<std::string>(&v))
        std::cout << "string: " << *s << '\n';
    else if (int* i = get<int>(&v))
        std::cout << "int: " << *i << '\n';
}
```

- Visitation

```
struct printVis : boost::static_visitor<> {
    void operator()(std::string s) const
    { std::cout << "string: " << s << '\n'; }
    void operator()(int i) const
    { std::cout << "int: " << i << '\n'; }
};

void printViaVis(V& v)
{ apply_visitor(printVis(), v); }
```

dts::variant

- Developed by Godbolt & Liber
  - Working at low latency trading firm
  - Annoyances with Boost.Variant
- Never want double buffering
  - Especially temporary heap backup
  - Almost always add boost::blank to the types
  - Visitors always have to handle boost::blank
- Want assignment to always be straightforward

dts::variant

- Developed by ***Matt*** Godbolt & Nevin ":-)" Liber
  - Working at low latency trading firm
  - Annoyances with Boost.Variant

- Never want double buffering
  - Especially temporary heap backup
  - Almost always add boost::blank to the types
  - Visitors always have to handle boost::blank
- Want assignment to always be straightforward

dts::variant

- “Empty” state: models **at-most-1**
- Assignment
  - Same type: operator=
  - Change type: Destroy+construct
    - If exception is thrown, empty state
- Comparisons: alternative-then-values
- Default constructible: empty state
- Element access: get<type> and visitation

dts::variant

```
dts::variant<A, B> v1 = A();
v1 = B();
```

dts::variant

```
dts::variant<A, B> v1 = A();
v1 = B();
```

dts::variant

- Very familiar with the issues involved
- Thought about proposing it
  - Open source (easy)
- Too many battle scars from failing to get optional into C++14
  - Constantly revisiting old arguments without new information
  - Comparison operators

std::variant v1

- First modern variant proposal
  - Axel Naumann
    - CERN
  - N4218
    - Urbana meeting (2014)
    - Targeting Library Fundamentals TS

std::variant v1

- Has an empty state
- Assignment: always destroy-then-construct
  - Even homogeneous assignment
- Comparisons: values
- Default constructible: empty state
- Element access: only get<type>

std::variant v1

- LEWG had 17 people for this discussion
- Alternative to empty state
  - Constraint types allowed in variant
- “I'm terrified of a world where everyone will invent their own visitor.”
  - No better alternative suggested
- Comparison of values
  - Full matrix of comparisons (O(n^2) at compile time)

std::variant v1

- Visitation
  - Bjarne Stroustrup
    - Visitation is unpleasant
    - Lots of requests for extension
  - Should just use pattern matching
    - Language feature
    - Not proposed
    - Not for Library Fundamentals TS
  - Shouldn’t wait for pattern matching
    - Axel agreed to trim visitation to the bare minimum for variant

std::variant v1

- However
  - Paper proposed a solution without exploring the design space
  - Asked Axel to do more work
    - Tuple-like interface
    - Constexpr
    - Never-empty
    - variant<int, int>, variant<int, const int>, etc.
    - Visitation

std::variant v2

- N4450 - pre-Lenexa (2015)
- Assignment
  - Uses operator= if the held type isn't changed

std::variant v2

- Alternatives to empty state
  - Only types that are noexcept copy constructible
    - Leaves out all types that do allocation
      - vector, string, etc.
- Double buffering
  - Construct-then-destroy
  - Double the memory

std::variant v2

- Comparisons by alternative-then-values
- Axel agreed because otherwise transitivity was broken
  - Still hoping for a way to get it

std::variant v2

- N4450 - pre-Lenexa mailing (2015)
  - *And then the committee woke up…*
  - Hundreds of emails
    - Many of which were mine
  - Heated, contentious technical discussion

std::variant v2

- Comparison suggestions
- Because short is comparable with int, variant<short, int> should not be comparable with itself because of possibly surprising results
  - O(n^2) at compile time

std::variant v2

```
variant<A, B> v1 = A();
v1 = B();
```

- Assignment suggestions
  - If noexcept move constructible
    - Construct temporary copy of B
      - If exception, v1 still has A
    - Destroy A in v1
    - Move construct temporary B into v1
    - Destroy temporary B

std::variant v2

```
variant<A, B> v1 = A();
v1 = B();
```

- Suggestion: Really want never-empty
  - Otherwise it isn’t quite a sum type

std::variant v2

```
variant<A, B> v1 = A();
v1 = B(); // throws
```

- Sean Parent
  - Instead of empty state, how about a partially formed state when an exception is thrown?
    - Only assignment to and destruction allowed
      - Assignment from not allowed
    - Cannot even query this state
    - Default construction is not this state
    - Partially formed is still a sum type

std::variant v2

```
variant<A, B> v1 = A();
v1 = B(); // throws
```

- Sean Parent on Partially Formed State *(cont.)*
    - By not requiring noexcept move operations, providing the basic exception guarantee becomes complicated
- Basic exception guarantee requirement, moved-from state and default constructible are all related
- Minimum we had to satisfy
- Fine with more functionality
    - Didn’t object to empty state

std::variant v2

```
variant<A, B> v1 = A();
v1 = B(); // throws
```

- Suggestion: Terminate on exception
  - User has no way to check ahead of time so as to avoid termination

std::variant v2

```
variant<A, B> v1 = A();
v1 = B(); // throws
```

- Suggestion: limited to types that are noexcept move constructible
    - Draconian
        - Eliminates most legacy (pre-C++11) types and aggregates of them
- Non-portable
    - list(list&&) not required to be noexcept
    - Types which aggregate list (use as member variable)

std::variant v2

```
variant<A, B> v1 = A();
v1 = B(); // throws
```

- Suggestion: limited to types that are noexcept move constructible
  - Expert only to determine for implicitly declared move constructors

```
struct Alpha {
    Bravo b;
    Charlie c;
    Delta d;
};
```

  - No visual cues

std::variant v2

```
variant<A, B> v1 = A();
v1 = B(); // throws
```

- Suggestion: disallow throwing move constructors in the language
  - It had been tried before
    - And failed…

std::variant v2

```
variant<A, B> v1 = A();
v1 = B(); // throws
```

- Suggestion: Make variant a built-in (as opposed to library) type
  - Nice for other reasons (more on that later), but doesn't address the problem

std::variant v2

```
variant<A, B> v1 = A();
v1 = B(); // throws
```

- Suggestion: Add a strong_assign() member function to types which doesn't throw
  - People will still just use operator=

std::variant v2

```
variant<A, B> v1 = A();
v1 = B(); // throws
```

- Suggestion: Undefined behavior
  - Bad as termination
  - Users have no way to avoid it

std::variant v2

```
variant<A, B> v1 = A();
v1 = B(); // throws
```

- Suggestion: Multiple variant types standardized
  - Only experts can choose between them
  - Interoperability between them

std::variant v2

```
variant<A, B> v1 = A();
v1 = B(); // throws
```

- Summary of the top ideas:
  - Double buffering
  - Empty or partially formed state
  - Restrict the types
  - Terminate / undefined behavior
- About 1/3 of my email messages kept repeating this to keep the discussion focused

Partially formed state

- Only destroy or assign to it
  - Cannot query if variant is in this state
- Should a default constructed variant be that state?

Partially formed state

- Alisdair Meredith

- Problematic: hidden unexpected pre-condition on moving/copying

```
std::vector<variant<A>> vv(1);
vv.reserve(vv.capacity() + 1); //internally moves/copies
```

- Most types have no preconditions on moving/copying

Sidebar

- Containers are value initialized when default constructing elements
    - Fundamental types are zero initialized
- Developers want a way to default initialize a container (especially contiguous containers string & vector)
    - Default initialization for fundamental types is uninitialized memory

```
std::vector<float, default_init_allocator> vf(1);
vf.reserve(vf.capacity() + 1);
```

    - Even assuming copying uninitialized floats is defined behavior
        - What happens if the uninitialized memory happens to be the bit pattern for signaling_NaN?
    - Hard to convince me to standardize this
    - Hard to convince committee to standardize this
- Copying dangling pointers has the same issue
    - Such as implementations of weak_ptr

Sidebar to the Sidebar

```
std::vector<float, default_init_allocator> vf(1);
vf.reserve(vf.capacity() + 1);
```

- What if we just allocate an array of default initialized values?
    - Eliminates the accidental copying problem

```
std::make_unique_default_init<float[]>(2);
```

- In the C++20 CD (Committee Draft)
- As of last week, In C++20 *hopefully*…
    - Name is contentious
    - Should it be constrained to trivially default constructible types?
- One of two NB comments that didn't get resolved

std::variant v2

- Discussed in Lenexa (2015)
- Invited all interested parties
  - Unfortunately, the committee members who don’t normally attend LEWG couldn’t make it (too much to do at committee meetings)
- I strongly suggested an evening session, but to no avail
- “There’s never enough time to do it right, but there's always enough time to do it over.” - Jack Bergman

std::variant v2

- Goal: answer all open questions
- Small subgroup of LEWG (7 people) - not contentious
  - Emptiness vs nonemptyness
  - Assignment between heterogeneous variants
  - Can types be repeated
  - int vs. const int
  - Allow references?
  - Allow void?
  - Binary relation operators between heterogeneous variants?
  - Default construction: empty state or alternative zero type or not at all?
  - How do we define or deduce the return type of visitor?

std::variant v2

- LEWG discussion (17 people) - not contentious
  - Polls (round 1)
    - *Strongly favor - favor - neutral - against - strongly against*
    - Query empty state: 4-4-4-1-1
    - Default constructor in empty state: 3-1-3-1-5
    - Default constructor try alternative zero type: 5-3-1-2-2

std::variant v2

- More discussion after lunch
  - Heterogeneous assignment
- Is variant<short, int> assignable from variant<int, short>?
  - Checking every assignment is an O(n^2) problem at compile time
  - What if we could sort types?

Sorting types

- I was 99% sure we cannot sort types
  - No total ordering on types
    - Dynamic libraries

- Decided to informally ask CWG (Core Working Group) about this at breakfast the next day
  - Asked them not to laugh
    - So they cringed :-)
  - Confirmed my suspicions

std::variant v2

- Polls (round 2)
  - Remove heterogeneous (different variants) assignment: 9-5-3-0-1
- Conversions: 5-4-1-1-0
  - variant<int, std::string> = “Hello”;
- Heterogeneous comparison: 0-2-5-3-3
  - variant<int>(1) == 1
- variant<A, B, C> == variant<X, Y, Z>: 0-1-0-4-8

std::variant v2

- Polls (round 2)
  - variant<int, int> allowed?
    - Model disjoint union: 14
      - Ordered sequence of types, identified by alternative
        - For equal types, pick first: 3
        - For equal types, pick neither by type (not polled)
    - Non-disjoint union:
      - Unordered sequence of types
- variant<int, const int>: 9-4-1-1-1
- variant<>: Valid type that cannot be constructed

std::variant v2

- Polls (round 2)
- Allow types to be reference types: 6-4-6-1-0
  - Note: optional does not support reference types
- Allow void: 6-9-2-0-0

std::variant v2

- Polls (round 2)
  - Multi visitation: 0-7-7-1-0
  - Binary visitation: 0-1-10-1-3
  - Visitor return type
    - common_type: 12
      - Note: this is order-dependent
    - Same return type: 13
    - Return type of operator()() (empty state): 1
    - Return variant<return types…>: 2
    - Return variant<return types…> if distinct, otherwise single return type: 0

std::variant v2

- Polls (round 2)
- Pointer version of get(variant<…>*): unanimous consent
  - Similar to any_cast and dynamic_cast

std::variant v2

- Polls (round 2)
- Default construct into empty state: 2-0-2-1-6
- Default construct alternative 0 type: 6-3-0-1-1
- Default construct first default constructible type: 0-1-2-5-3

std::variant v2

- Polls (round 2)
  - index() return -1 on empty: 4-1-3-1-2
  - visit/get has precondition that variant not empty: 4-8-2-0-0
- Type list and utilties: NO
  - Separate proposal as it is needed more generally
- Access to underlying data buffer: NO

std::variant v3

- N4516 - written in Lenexa
  - Empty state
    - Only when assignment fails
  - Assignment
    - Same type: operator=
    - Failure changing types: empty state
    - Accidentally missing v = A(); (assignment from exact types if that type only appears once in the type list)
- Comparison: alternative-then-values
  - Only comparisons for variants of the same type
- Default construction: first alternative type
- Element access: get and visitor

std::variant v3

- LEWG Discussion Lenexa (Friday - 16 people)
  - One person still wants to default construct initialize to empty state. Nobody else wants that
  - Separate way to query empty state: valid()
  - Polls (round 3):
    - If valid(), index() return magic value: 5-3-1-2-2
    - index() has a precondition of valid(): 5-2-0-3-3
    - Neither is consensus; paper need not change

std::variant v4

- N4542 - post-Lenexa mailing
- Empty state when assignment fails
- visit() has variadic signature (multiple heterogeneous variants)

std::variant v4

- Assignment when switching types changed to minimize chances of entering invalid state
    - But not necessarily eliminate it!
        - Copy right hand side contents to temporary
        - Destroy left hand side contents
        - Move constructs left hand side from temporary
- Copy-then-destroy-then-move-then-destroy
    - Pessimization!
    - Probabilistic argument
        - Evidence?

Pessimization

- Generally speaking, one should not make claims about optimization/pessimization without measuring
  - Humans have a hard time reasoning performance when doing different work
  - However, in this case we are strictly doing more work
    - Also, assumes move is always an optimization of copy
      - variant<array<int, 1000>, array<double, 1000>>
- Still, one should measure…

std::variant v4

- And the committee was woken once again…
  - At least 1,080 more email messages…
    - Half of which showed up the week after the mailing
      - Many by the people who could have but didn't attend the LEWG sessions and didn't like the consensus
      - Every single point brought up before was brought up again
      - 2/3 of my emails contained "But this is not new information…"
- There were a few more ad-hoc suggestions…

More suggestions

- The group that evaluated it was too small
- Make is so the folks at C++Now like it
- Head of a National Body delegation: Don’t even think about putting it into the standard before putting it into a TS
- Make it so the active type cannot be changed
  - Use optional<variant<…>> if you want to change the active type
- Make variant move-only
- Transactional memory

std::variant v5

- P0080R0 pre-Kona
  - valid() is a visible state for get/visit
  - valid() no longer a precondition for copy/move from a variant

Competing papers

- P0087R0 (also by Axel) - std::variant 2.1
- Default construction into the empty state
- Assignment: copy+destroy+move
- No comparisons

Competing papers

- P0080 - “Variant: Discriminated Union with Value Semantics” - Michael Park
  - Indeterminate state
    - Default construction
    - Assignment failure

- Assignment: Destroy+Construct, unless noexcept move constructible, then Copy/Move+Destroy+Construct
- type_switch instead of visit
  - Precursor to pattern matching
- Never presented (logistical reasons getting to Kona)

Competing papers

- David Sankel
  - P0092R0 - “Simply a basic variant”
    - Basic exception guarantee
      - Double buffering unless one of the types is noexcept default constructible

Competing papers

- David Sankel
  - P0093R0 - “Simply a strong variant”
    - Strong exception safety guarantee
    - Never-empty
      - Double buffering

Competing papers

- David Sankel
  - P0094R0 - “The Case for a Language Based Variant”
    - Alternatives have names

```
enum union command {
    std::size_t set_score; // Set the score to the specified value
    std::monotype fire_missile; // Fire a missile
    unsigned fire_laser; // Fire a laser with the specified intensity
    double rotate; // Rotate the ship by the specified degrees.
};
```

    - Pattern matching instead of visitation

```
switch( cmd ) {
    case set_score value:
        stream << "Set the score to " << value << ".\n";
    case fire_missile m:
        stream << "Fire a missile.\n";
    case fire_laser intensity:
        stream << "Fire a laser with " << intensity << " intensity.\n";
    case rotate degrees:
        stream << "Rotate by " << degrees << " degrees.\n";
}
```

Kona 2015

- Evening session (paraphrasing): “If I hear repeated arguments, I'll call it out and stop minuting.” - scribe
    - They only had to do this once
- P0088R0 std::variant v5
    - Exceptions on invalid state operations (valid() never a precondition so no undefined behavior): 13-15-2-3-0
        - That’s all that had to change…
- P0094R1 enum union and pattern matching
    - Encourage the author to do more work

Undefined Behavior

- Precondition violation
  - Undefined behavior
    - No-false-positive sanitizers can detect bugs
    - Optimizers can optimize assuming it never happens
    - vector::operator[](n) where n >= size()

Undefined Behavior

- Exceptions
  - No undefined behavior
  - All states are valid
  - Cannot tell the difference between accidental use (bug) and deliberate use (feature)
  - vector::at()

Undefined Behavior

- Which is better?
  - It depends
  - Even defining reasonable behavior can be error prone
    - unsigned arithmetic wraps
      - No undefined behavior
    - signed arithmetic underflow/overflow
      - Undefined behavior

Undefined Behavior

- Back to variant
  - When !valid(), what operations should be errors?
- The committee struggles with this
- Contracts (C++~~20~~23?) gives us the language and framework to describe this

Kona 2015

- LEWG discussion (19 people)
- Allow conversion (if unique) for both construction and assignment: 4-4-3-4-0
- Bike shedding
  - valid() —> !valueless_by_exception()

std::variant v6

- P0088R1

- Incorporate Kona changes

- Bug fixes

std::variant v7

- P0088R2
  - Instead of targeting a TS, ***target C++17!***
  - After the Jacksonville 2016 meeting (v6 discussed), people were asked if we could possibly gather consensus in Oulu to target C++17. This would be the last chance before feature freeze.
  - General feeling: we have consensus; ship it!

std::variant v7

- P0088R2 - Axel Neumann
  - Introduction by David Sankel
    - “C++***17*** needs a type-safe union:
    - “Lets not make the same mistake we made with `std::optional` by putting this library into a TS. We waited three years where no substantial feedback or discussion occurred, and then moved it into the IS virtually unchanged. Meanwhile, the C++ community suffered, and we continue to suffer from lack of this essential vocabulary type in interfaces.
    - “The implications of the consensus `variant` design are well understood and have been explored over several LEWG discussions, over a thousand emails, a joint LEWG/EWG session, and not to mention 12 years of experience with Boost and other libraries. The last major change made to the proposal was non-breaking and added exception throws where previously there was undefined behavior. Since then, all suggested modifications have been cosmetic, rehashes of older discussions, or would be handled just as well by defect resolutions.
    - “The C++ community should not wait three years for a widely useful library that is already done, fits its purpose, and has had such extensive review. There is a low chance that we will regret including variant in C++17, but a high chance that we will regret omitting it.”

Competing paper

- P0308 - “Valueless Variants Considered Harmful” - Peter Dimov

  - Pilfering constructor
    - For types with a noexcept(false) move constructor
    - Destructive move
      - Object can be destroyed, but nothing else

Oulu 2015

- P0088R3 - std::variant v8
  - Bug fixes
- Final plenary
  - NB (paraphrased) - “We will not accept std::any in 17 without std::variant. People will use the wrong type”
  - “Move we apply to the C++ Working Paper the Proposed Wording from P0088R3, Variant: a type-safe union for C++17”
    - ***Unanimous consent!***

Issaquah 2016

- P0510R0 - “Disallowing references, incomplete types, arrays, and empty variants” - Erich Keane
  - In response to national body comments, variant can no longer store references, incomplete types (including void) and arrays
    - Optional also doesn’t support references
    - Cannot use incomplete types because variant needs to know the size of the type
    - arrays are weird
- variant<> can no longer be instantiated

std::variant C++17

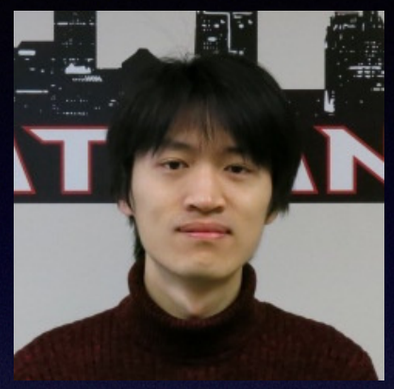
- Empty state
  - valueless_by_exception
- Assignment
  - Same type
    - operator=
  - Change type
    - Copy assignment
      - Copy-then-destroy-then-move-then-destroy
        - *(Pessimization)*
    - Move assignment
      - Move-then-destroy
- Comparisons
  - Alternative-then-values
  - Homogeneous
    - variant<A, B> only comparable with variant<A, B>
- Default construction
  - Alternative 0 type
- Element access
  - get
  - visit

Let's Brainstorm!

```
variant<A, B> v1 = A();
v1 = B();
```

- `v1` holds an object of type `A`
- We want to store an object of type `B`
- Constructor for `B` throws
- ***How did we do?***

std::variant C++CD (N4835)



- Zhihao Yuan
  - P0602 variant and optional should propagate copy/move triviality
  - Added constexpr to copy/move
  - If all held types are trivially copy/move constructible/assignable, then variant is trivially copy/move constructible/assignable

std::variant C++CD (N4835)

- P0608 A sane variant converting constructor (Zhihao Yuan)
  - Unintended alternatives fix

```
variant<string, bool> x = "abc";                // holds bool
```

  - Information loss fix

```
variant<char, optional<char16_t>> x = u'\u2043';  // holds char = 'C'
double d = 3.14;
variant<int, reference_wrapper<double>> y = d;    // holds int = 3
```

  - Unstable construction fix

std::variant C++CD (N4835)

- LEWG 3228 (Barry Revzin)

```
struct ConvertibleToBool
{
  constexpr operator bool() const { return true; }
};
```

```
static_assert(std::holds_alternative<bool>(std::variant<int, bool>(ConvertibleToBool{})));
```

- Before P0608, holds bool
- After P0608, holds int
- Proposed fix P1957 (Zhihao Yuan): core language change
  - Conversion from pointer (or pointer to member) to bool becomes a narrowing conversion

optional and variant

- Is optional<T> a refinement of variant<nullopt_t, T>?
  - Not quite
    - optional<T> has no valueless_by_exception state
    - Assignment
      - Same type:
        - operator=
      - Change type
        - Destroy-or-construct a T
          - Degenerate case of destroy+construct
          - No extra move pessimization
- No support for get or visit
  - But could be added

Odds and Ends

- Given that `float` and `int` are trivially noexcept move/copy constructible

- Can `variant<float, int>` get into the `valueless_by_exception` state?

  - Sadly, yes…

Odds and Ends

- Agustín Bergé
  - Eggs.Variant (2014)

```
struct S { operator int() { throw 42; } };
//…
    variant<float, int> v{12.f};
    try { v.emplace<1>(S()); }
    catch(...){}
    assert(v.valueless_by_exception());
```

- emplace<1> to emplace an int
  - Destroy 12.f
  - Call S::operator int()
    - throws

Proposed Language Variant

- P0095R2 - Language Variants (David Sankel, Dan Sarginson, Sergei Murzin)
  - Ivariant keyword
    - Ivariant is to variant as struct is to tuple
    - Alternatives have names
    - Default constructible only if first type is default constructible
    - Inspection
      - inspect keyword (pattern matching)
    - Assignment
      - If any of the types can throw during move construction or assignment, then no defaulted assignment operator
        - Users can implement their own

Proposed Language Variant

- Discussed in EWGI at Kona 2019
  - Much rehash of std::variant design
    - Can still get into valueless_by_exception state
    - Should that be queryable?
    - What about making that state UB?
- What about throwing destructors?
  - Language feature has to account for it
- Assignment
  - What about =default?
    - If so, we should do it for classes
  - No emplace proposed yet

Boost.Variant2

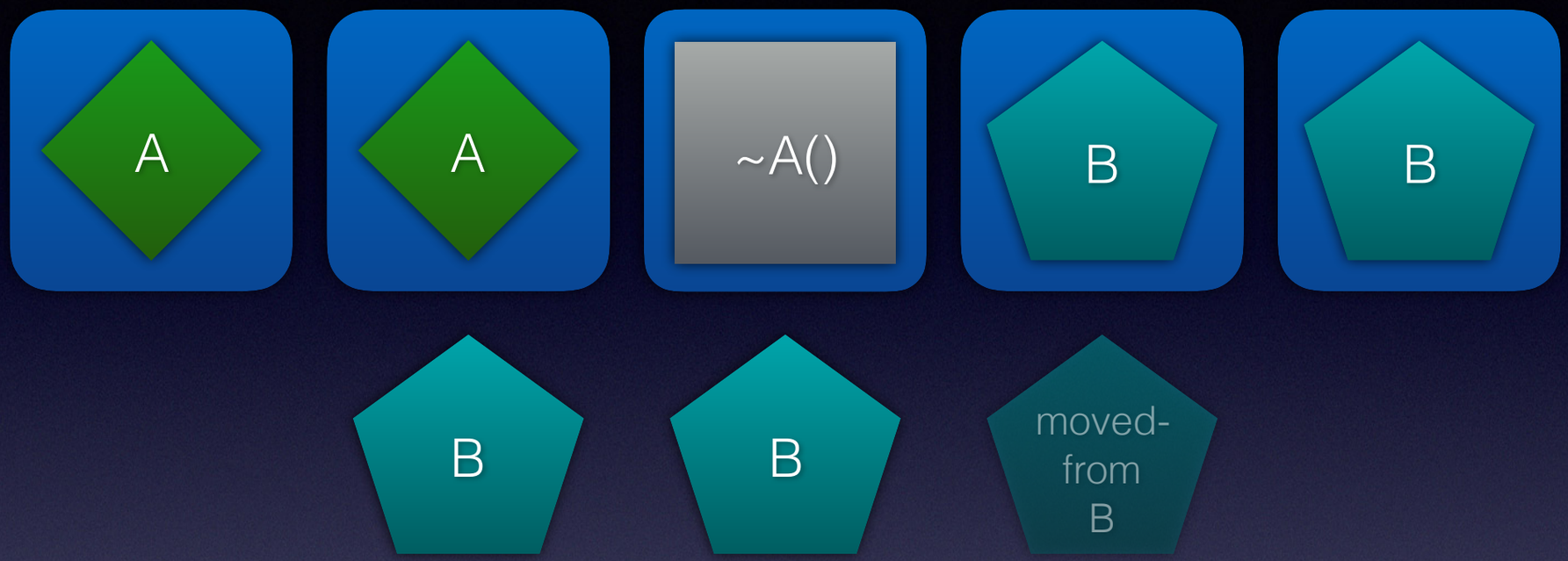
- Peter Dimov
- Never-empty guarantee discussion
    - If all types have non throwing assignment, the variant cannot fail during assignment
    - Otherwise, if an explicit null state is available, that state is chosen on assignment failure
    - Otherwise, if at least one type is noexcept default constructible, that state is chosen on assignment failure
    - Otherwise, double buffering
- Hundreds of emails…
    - *Sound familiar?*

Boost.Variant2

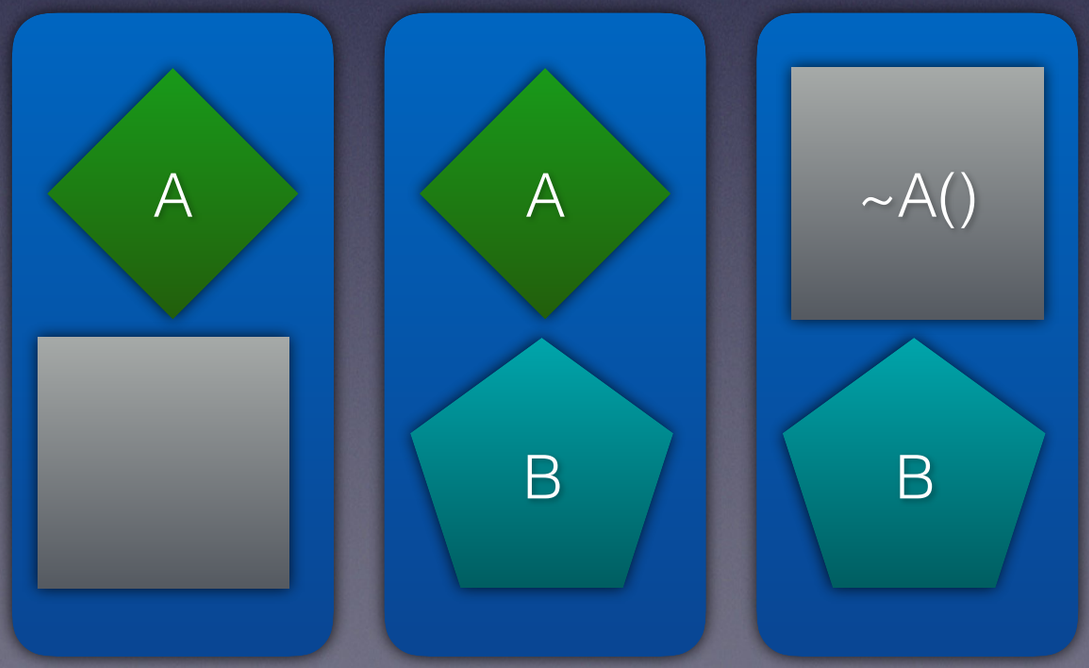
- Never-empty
- Strong exception safety guarantee

Boost.Variant2

All types have a non-throwing move constructor



or



Double Buffering

Pattern Matching

- P1371R1 Sergei Murzin, Michael Park, David Sankel, Dan Sarginson
  - Discussed in Cologne (2019) post-meeting and Belfast (2019)
  - Before

```
struct visitor {
  void operator()(int i) const {
    os << "got int: " << i;
  }
  void operator()(float f) const {
    os << "got float: " << f;
}
  std::ostream& os;
};
std::visit(visitor{strm}, v);
```

  - After

```
 inspect (v) {
  <int> i: strm << "got int: " << i;
  <float> f: strm << "got float: " << f;
}
```

Nevin variant

- Wanted a variant to replace usage of Boost.variant
    - std::variant is that variant
    - std::variant is good
- Two other variants
    - Strong performance
        - dts::variant + refinement of optional
        - Always construct/assign directly into the variant
        - Never construct-then-destroy-then-move to avoid exceptions
    - Strong reasoning
        - Double buffering within the variant
        - Construct-then-destroy (not sure what assignment should do)
        - Strong exception safety guarantee
        - Never-empty guarantee

The Many Variants of `std::variant`

- Special Thanks (thank them / blame me)
  - My family
  - My various employers
  - The C++ Community
  - The C++ Committee
  - David Sankel, Sean Parent, Michael Park & Matt Calabrese
  - *And especially…*

The Many Variants of `std::variant`

- Axel Naumann

- For championing std::variant

- ***“Thank you, Nevin “:-)” Liber, for bringing sanity to this proposal”***

Q&A

Argonne NATIONAL LABORATORY