Variant: Discriminated Union with Value Semantics

Document #:	P0080
Date:	2015-07-28
Project:	Programming Language C++ Library Evolution Group

Reply-to:	Michael Park <mcypark@gmail.com>

Contents

1 Introduction

This paper is an alternative proposal to [N4542], which proposes a design for a discriminated union. This topic has been discussed **extensively** within the ISO C++ Standards Committee as well as **std-proposals** mailing list. The goal of this paper is to identify the concerns of the polarizing groups, and to propose a solution that would address them with minimal compromise.

2 Motivation and Scope

Due to the lack of discriminated union with value semantics in C++, they are typically handled in one of 3 ways:

- `union`-like class, with an `enum` as the discriminator
- Class inheritance hierarchy
- `enum`, in the special case where all alternatives are simply unit types.

2.1 `union`-like Class

A `union`-like class typically consists of the `union` which contains one of the possible alternatives, and an `enum` that represents the discriminator.

```
struct Token {
  Token(int i) : int_(i), type(Int) {}
  Token(std::string s) : string_(std::move(s)), type(String) {}

  ~Token() {
    switch (type) {
      case Int:
        break;
      case String:
        using std::string;
        string_.~string();
        break;
    }
  }

  enum { Int, String } type;
  union {
    int int_;
    std::string string_;
  };
};
```

Note that after all of this code, this class still only supports construction and destruction. Even more code would need to be written to support other operations such as element access, assignment, visitation, etc.

Furthermore, it still only handles exactly one set of types: `int` and `std::string`. A whole new class definition would need to be written for a different set of types consisting of more or less the exact same code.

2.2 Class Inheritance Hierarchy

A class inheritance is a discriminated union in the sense that a pointer to an abstract base class can only be pointing to an instance of one of the derived classes.

```cpp
struct Token {};
struct Int : Token { int value; };
struct String : Token { std::string value; };
```

There are several disadvantages to this approach:

- `Token` no longer has value semantics. It must be passed around by pointer or reference in order to avoid object slicing.
- `Int`, and `String` classes needed to be introduced since `int` and `std::string` cannot inherit from `Token`.
- If a type needs to be a member of multiple discriminated unions, multiple inheritance needs to be introduced.
- The use of virtual functions typically scatters the individual cases for a single algorithm into different parts of the codebase which becomes an engineering burden. The visitor pattern is a popular solution to this problem and also makes it possible for multi-visitation, but it introduces non-trivial amount of boilerplate code only to support exactly one set of types.

2.3 `enum`

In the special case where the members of a discriminated union do not have associated data, an `enum` is typically used to achieve the behavior.

```cpp
enum class Color { Blue, Green, Red };
```

This paper proposes a library solution for a generic, type-safe discriminated union with value semantics with the class template `variant<Types...>`.

The following is an example of how the library may be used:

```cpp
#include <iostream>
#include <string>
#include <variant>

struct custom_equal {
  template <class T>
  bool operator()(const T& lhs, const T& rhs) const { return lhs == rhs; }

  template <class T, class U>
  bool operator()(const T&, const U&) const { return false; }
};

int main() {
  using namespace std::string_literals;

  // direct initialization
  std::variant<int, std::string> v("hello world!"s);

  // direct access
  const std::string& s = std::get<std::string>(v);
```

```cpp
  assert(s == "hello world!"s);

  try {
    int i = std::get<int>(v);  // throws: std::bad_variant_access
  } catch (const std::bad_variant_access& ex) {
    // handle exception
  }

  // copy (and move) construction
  std::variant<int, std::string> w(v);

  // assignment
  try {
    v = 42;
  } catch (const std::bad_variant_assign& ex) {
    // ‘v‘ is in an indeterminate state. The exception thrown by the move
    // constructor is available as ‘ex.nested_ptr‘.
  } catch (const std::exception& ex) {
    // v is valid.
  }

  // visitation
  std::type_switch (v) (
    [](const auto& value) { std::cout << value; }
  );  // prints: 42

  std::type_switch (w) (
    [](int value               ) { std::cout << "int: "    << value; },
    [](const std::string& value) { std::cout << "string: " << value; }
  );  // prints: string: hello world!

  bool result = std::type_switch (v, w) (custom_equal{});
  assert(!result);

  std::type_switch (v, w) (
    [](int               , int               ) { std::cout << "(int, int)"      ; },
    [](int               , const std::string&) { std::cout << "(int, string)"   ; },
    [](const std::string&, int               ) { std::cout << "(string, int)"   ; },
    [](const std::string&, const std::string&) { std::cout << "(string, string)"; }
  );  // prints: (int, string)
```

The support for discriminated unions are common in many other languages such as Haskell, ML, Rust, Swift, and F#.

3 Impact on the Standard

This proposal is a pure library extension, and does not require any new language features.

4 Proposed Wording

Make the following changes to the experimental `<optional>` header:

```
namespace std {
-  constexpr in_place_t in_place{};
+  in_place_t in_place() { return {}; }

-  template <class... Args> constexpr explicit optional(in_place_t, Args&&...);
+  template <class... Args> constexpr explicit optional(in_place_t (&)(), Args&&...);

   template <class U, class... Args>
-  constexpr explicit optional(in_place_t, initializer_list<U>, Args&&...);
+  constexpr explicit optional(in_place_t (&)(), initializer_list<U>, Args&&...);
}
```

Add a new subclause `20.N Variants` under `20 General utilities library:`

20.N Variants [variant]

20.N.1 In general [variant.general]

This subclause describes the `variant` library that provides a discriminated union as the class template `variant`. Each template argument specifies the type of an element that can be stored in the `variant` object.

Header `<variant>` synopsis:

```
namespace std {
  // 20.N.2 class template variant:
  template <class... Types> class variant;

  // 20.N.3 class bad_variant_access:
  class bad_variant_access;

  // 20.N.4 class bad_variant_assign:
  class bad_variant_assign;

  // 20.N.5 in-place construction:
  template <size_t I> in_place_t in_place(integral_constant<size_t, I>) { return {}; }
  template <class T> in_place_t in_place(T) { return {}; }

  // 20.N.6 nullvar:
  struct nullvar_t {};
  constexpr nullvar_t nullvar{};

  // 20.N.7 element access:
  template <size_t I, class... Types>
  constexpr remove_reference_t<tuple_element_t<I, tuple<Types...>>>* get(
      variant<Types...>*) noexcept;

  template <size_t I, class... Types>
  constexpr remove_reference_t<const tuple_element_t<I, tuple<Types...>>>* get(
```

```
    const variant<Types...>*) noexcept;

template <class T, class... Types>  // only if see below
constexpr remove_reference_t<T>* get(variant<Types...>*) noexcept;

template <class T, class... Types>  // only if see below
constexpr remove_reference_t<const T>* get(const variant<Types...>*) noexcept;

template <size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>& get(variant<Types...>&);

template <size_t I, class... Types>
constexpr const tuple_element_t<I, tuple<Types...>>& get(const variant<Types...>&);

template <size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>&& get(variant<Types...>&&);

template <size_t I, class... Types>
constexpr const tuple_element_t<I, tuple<Types...>>&& get(const variant<Types...>&&);

template <class T, class... Types>
constexpr T& get(variant<Types...>&);  // only if see below

template <class T, class... Types>
constexpr const T& get(const variant<Types...>&);  // only if see below

template <class T, class... Types>
constexpr T&& get(variant<Types...>&&);  // only if see below

template <class T, class... Types>
constexpr const T&& get(const variant<Types...>&&);  // only if see below

// 20.N.8 relational operators:
constexpr bool operator==(nullvar_t, nullvar_t) { return true;  }
constexpr bool operator!=(nullvar_t, nullvar_t) { return false; }
constexpr bool operator< (nullvar_t, nullvar_t) { return false; }
constexpr bool operator> (nullvar_t, nullvar_t) { return false; }
constexpr bool operator<=(nullvar_t, nullvar_t) { return true;  }
constexpr bool operator>=(nullvar_t, nullvar_t) { return true;  }

template <class... Types>
constexpr bool operator==(const variant<Types...>&, const variant<Types...>&);

template <class... Types>
constexpr bool operator!=(const variant<Types...>&lhs,
                          const variant<Types...>&rhs) { return !(lhs == rhs); }

template <class... Types>
constexpr bool operator<(const variant<Types...>&, const variant<Types...>&);

template <class... Types>
constexpr bool operator>(const variant<Types...>&lhs,
                         const variant<Types...>&rhs) { return rhs < lhs; }

template <class... Types>
constexpr bool operator<=(const variant<Types...>&lhs,
```

```
                          const variant<Types...>&rhs) { return !(lhs > rhs); }

  template <class... Types>
  constexpr bool operator>=(const variant<Types...>&lhs,
                          const variant<Types...>&rhs) { return !(lhs < rhs); }


  // 20.N.9 specialized algorithms:
  template <class... Types>
  void swap(variant<Types...>& lhs,
            variant<Types...>& rhs) noexcept(noexcept(lhs.swap(rhs)));

  // 20.N.10 hash support:
  template <> struct hash<nullvar_t>;
  template <class... Types> struct hash<variant<Types...>>;

  // 20.N.11 visitation:
  template <class R = unspecified:deduce_tag, class... Variants>
  unspecified:TypeSwitch<R, Variants...> type_switch(Variants&&...);
}
```

20.N.2 Class template **variant** [variant.variant]

```
namespace std {
  template <class... Types>
  class variant {
  public:
    // 20.N.2.1 variant construction:
    constexpr variant() noexcept;

    template <size_t I, class... Args>
    explicit constexpr variant(in_place_t (&)(integral_constant<size_t, I>),
                               Args&&...);

    template <size_t I, class U, class... Args>  // only if see below
    explicit constexpr variant(in_place_t (&)(integral_constant<size_t, I>),
                               initializer_list<U>,
                               Args&&...);

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


    template <class T, class U, class... Args>  // only if see below
    explicit constexpr variant(in_place_t (&)(T), initializer_list<U>, Args&&...);

    template <class U> constexpr variant(U&&);  // only if see below

    variant(const variant&);
    variant(variant&&) noexcept(see below);

    // 20.N.2.2 variant destruction:
    ~variant();

    // 20.N.2.3 variant assignment:
```

```
    template <size_t I, class... Args> void emplace(Args&&...);

    template <size_t I, class U, class... Args>
    void emplace(initializer_list<U>, Args&&...);  // only if see below

    template <class T, class... Args> void emplace(Args&&...);

    template <class T, class U, class... Args>
    void emplace(initializer_list<U>, Args&&...);  // only if see below

    template <class U> variant& operator=(U&&);  // only if see below

    variant& operator=(const variant&);
    variant& operator=(variant&&) noexcept(see below);

    // 20.N.2.4 variant observers:
    constexpr int index() const noexcept;
    constexpr const type_info& type() const noexcept;

    // 20.N.2.5 variant swap:
    void swap(variant&) noexcept(see below);
  };
}
```

20.N.2.1 Construction [variant.cnstr]

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

```
constexpr variant() noexcept;
```

Effects: Constructs the `variant` object in an indeterminate state.

```
template <size_t I, class... Args>
explicit constexpr variant(in_place_t (&)(integral_constant<size_t, I>),
                           Args&&... args);
```

Let `T` be the type `tuple_element_t<I, tuple<Types...>>`.

Requires: `I < sizeof...(Types) && is_constructible_v<T, Args&&...>` is `true`

Effects: Initializes the contained value of the `variant` as if constructing an object of type `T` with `forward<Args>(args)...`.

Postcondition: `index() == I`

Throws: Any exception thrown by the selected constructor of `T`.

Remarks: If the selected constructor of `T` is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

```
template <size_t I, class U, class... Args>  // only if see below
explicit constexpr variant(in_place_t (&ip)(integral_constant<size_t, I>),
                           initializer_list<U> init,
```

```
                        Args&&... args);
```

Effects: Equivalent to

```
template <size_t I, class... Args>
explicit constexpr variant(
    in_place_t (&)(integral_constant<size_t, I>), Args&&...);
```

with arguments `ip, init, forward<Args>(args)...`

Remarks: This constructor shall not participate in overload resolution unless `is_constructible_v< tuple_element_t<I, tuple<Types...>>, initializer_list<U>&, Args&&...>` is `true`. If the selected constructor of `T` is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

```
template <class T, class... Args>
explicit constexpr variant(in_place_t (&)(T), Args&&... args);
```

Let `I` be the index of `T` in `Types...`

Requires: `T` is in `Types...` && `T` is unique within `Types...`.

Effects: Equivalent to `variant(in_place<I>, forward<Args>(args)...);`

Remarks: If the selected constructor of `T` is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

```
template <class T, class U, class... Args>  // only if see below
explicit constexpr variant(in_place_t (&ip)(T),
                        initializer_list<U> init,
                        Args&&... args);
```

Effects: Equivalent to

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

with arguments `ip, init, forward<Args>(args)...`

Remarks: This constructor shall not participate in overload resolution unless `is_constructible_v<T, initializer_list<U>&, Args&&...>` is `true`. If the selected constructor of `T` is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

```
template <class U>
constexpr variant(U&& u);  // only if see below
```

Let `T` be one of the types in `Types...` for which `U&&` is unambiguously convertible to by standard overload resolution rules.

Effects: Equivalent to `variant(in_place<T>, forward<U>(u));`

Remarks: This constructor shall not participate in overload resolution unless there is a type `T` in `Types...` for which `U&&` is unambiguously convertible to by standard overload resolution

rules. If the selected constructor of `T` is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

```
variant(const variant& that);
```

Let `I` be the constant expression of `that.index()`.

Let `T` be `tuple_element_t<I, tuple<Types...>>`.

Requires: `is_copy_constructible_v<U>` is `true` for all `U` in `Types...`

Effects: Initializes the contained value of the `variant` as if constructing an object of type `T` with `get<I>(that)`.

Postcondition: `index() == I`

Throws: Any exception thrown by the selected constructor of `T`.

```
variant(variant&& that) noexcept(see below);
```

Let `I` be the constant expression of `that.index()`

Let `T` be `tuple_element_t<I, tuple<Types...>>`

Requires: `is_move_constructible_v<U>` is `true` for all `U` in `Types...`

Effects: Initializes the contained value of the `variant` as if constructing an object of type `T` with `get<I>(std::move(that))`.

Postcondition: `index() == I`

Throws: Any exception thrown by the selected constructor of `T`.

Remarks: The expression inside `noexcept` is equivalent to the logical `AND` of `is_nothrow_move_constructible_v<U>` for all `U` in `Types...`

20.N.2.2 Destruction [variant.dtor]

```
~variant();
```

Let `I` be the constant expression of `index()`

Let `T` be `tuple_element_t<I, tuple<Types...>>`

Effects: If the `variant` is initialized, calls `get<I>(*this).~T();`

20.N.2.3 Assignment [variant.assign]

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

```
template <size_t I, class... Args> void emplace(Args&&...);
```

Let `T` be the type `tuple_element_t<I, tuple<Types...>>`.

Requires: `is_constructible_v<T, Args&&...>` is `true`

Effects: Destructs the value in `*this`, then initializes the contained value of `*this` as if constructing an object of type `T` with the arguments `forward<Args>(args)...`

Postcondition: `index() == I`

Throws: `bad_variant_assign` or any exception thrown by the selected constructor of `T`.

Exception Safety: If an exception `E` is thrown during the initialization of the value in `*this`, `E` is caught and `bad_variant_assign` is thrown with `E` nested within it. In this case, no initialization takes place and `variant` is left in an indeterminate state. Otherwise, there are no effects.

```
template <size_t I, class U, class... Args>
void emplace(initializer_list<U> init, Args&&... args);  // only if see below
```

Effects: Equivalent to

```
template <size_t I, class... Args> void emplace(Args&&...);
```

with arguments `init, forward<Args>(args)...`

Remarks: Let `T` be the type `tuple_element_t<I, tuple<Types...>>`. This function shall not participate in overload resolution unless `is_constructible_v<T, initializer_list<U>&, Args&&...>` is `true`.

```
template <class T, class... Args> void emplace(Args&&...);
```

Let `I` be the index of `T` in `Types...`

Requires: `T` is in `Types...` && `T` is unique within `Types...`.

Effects: Equivalent to `emplace(in_place<I>, forward<Args>(args)...);`

```
template <class T, class U, class... Args>
void emplace(initializer_list<U> init, Args&&...);  // only if see below
```

Effects: Equivalent to

```
template <class T, class... Args> void emplace(Args&&...);
```

with arguments `init, forward<Args>(args)...`

Remarks: This constructor shall not participate in overload resolution unless `is_constructible_v<T, initializer_list<U>&, Args&&...>` is `true`.

```
template <class U> variant& operator=(U&&);  // only if see below
```

Let `T` be one of the types in `Types...` for which `U&&` is unambiguously convertible to by standard overload resolution rules.

Requires: `is_constructible_v<T, U&&>` is `true` && `is_assignable_v<T, U&&>` is `true`

Effects: If `*this` contains a value of type `T`, effectively performs

```
get<T>(*this) = forward<U>(u);
```

Otherwise, if `is_nothrow_constructible_v<T, U&&>` is `true`, performs

```
emplace<T>(forward<U>(u);
```

Otherwise, performs

```
emplace<T>(T(forward<U>(u));  // Note the temporary construction.
```

Returns: `*this`

Remarks: This constructor shall not participate in overload resolution unless there is a type `T` in `Types...` for which `U&&` is unambiguously convertible to by standard overload resolution rules.

```
variant& operator=(const variant& that);
```

Let `I` be the constant expression of `that.index()`

Let `T` be `tuple_element_t<I, tuple<Types...>>`

Requires: `is_copy_constructible_v<U>` is `true && is_copy_assignable_v<U>` for all `U` in `Types...`

Effects: Equivalent to `return *this = get<I>(that);`

Returns: `*this`

```
variant& operator=(variant&& that) noexcept(see below);
```

Let `I` be the constant expression of `that.index()`

Let `T` be `tuple_element_t<I, tuple<Types...>>`

Requires: `is_move_constructible_v<U>` is `true && is_move_assignable_v<U>` for all `U` in `Types...`

Effects: Equivalent to `return *this = get<I>(std::move(that));`

Returns: `*this`

Remarks: The expression inside `noexcept` is equivalent to the logical `AND` of `is_nothrow_move_constructible_v<U> && is_nothrow_move_assignable_v<U>` for all `U` in `Types...`

20.N.2.4 Observers [variant.observers]

```
constexpr int index() const noexcept;
```

Effects: Returns the index of the currently active alternative.

```
constexpr const type_info& type() const noexcept;
```

Let `I` be the constant expression of `index()`

Let `T` be `tuple_element_t<I, tuple<Types...>>`

Effects: Returns `typeid(T)`

20.N.2.5 Swap [variant.swap]

```
void swap(variant& that) noexcept(see below);
```

Let `I` be the constant expression of `index()`

Let `T` be `tuple_element_t<I, tuple<Types...>>`

Requires: `is_move_constructible_v<U>` is `true` and `U` shall satisfy the requirements of `Swappable` for all `U` in `Types...`

Effects: If `index() == that.index()`, performs

```
using std::swap;
swap(get<I>(*this), get<I>(that));
```

Otherwise, equivalent to calling

```
template <class T> void swap(T&, T&);
```

with arguments `*this, that`

Exception Safety: If an exception was thrown during the call to function `swap(get<I>(*this), get<I>(that))`, the state of the value of `*this` and that is determined by the exception safety guarantee of `swap` for `T`.

If an exception was thrown during the call to `swap(*this, that)`, the state of the value of `*this` and `that` is determined by the exception safety guarantee of `variant`'s move constructor and assignment operator.

Remarks: The expression inside `noexcept` is equivalent to the logical `AND` of `is_nothrow_move_constructible_v<U> && noexcept(iter_swap(declval<U*>(), declval<U*>))` for all `U` in `Types...`

20.N.3 Class `bad_variant_access` [variant.bad.variant.access]

```
namespace std {
  class bad_variant_access : public exception {
  public:
    explicit bad_variant_access();
    bad_variant_access(const bad_variant_access&) noexcept;
    bad_variant_access& operator=(const bad_variant_access&) noexcept;
    virtual const char* what() const noexcept;
  };
}
```

The class `bad_variant_access` defines the type of objects thrown as exceptions to report the situation where an invalid attempt is made to access the value of a `variant` via `get`.

Given an object `v` of type `variant<Types...>`,

- For index-based operations, `get<I>(v)` is invalid if `get<I>(&v) == nullptr`
- For type-based operations, `get<T>(v)` is invalid if `get<T>(&v) == nullptr`

```
explicit bad_variant_access();
```

Effects: Constructs an object of class `bad_variant_access`.

```
bad_variant_access(const bad_variant_access&) noexcept;
bad_variant_access& operator=(const bad_variant_access&) noexcept;
```

Effects: Copies an object of class `bad_variant_access`.

```
virtual const char* what() const noexcept;
```

Returns: An implementation-defined NTBS.

Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion and display as a `wstring` (21.3, 22.4.1.4).

20.N.4 Class `bad_variant_assign` [variant.bad.variant.assign]

```
namespace std {
  class bad_variant_assign : public exception, public nested_exception {
  public:
    bad_variant_assign(void*) noexcept;
    bad_variant_assign(const bad_variant_assign&) noexcept;
    bad_variant_assign& operator=(const bad_variant_assign&) noexcept;
    void* variant_ptr() const noexcept;
    virtual const char* what() const noexcept;
  };
}
```

The class `bad_variant_assign` defines the type of objects thrown as nested exceptions to report the situation where an exception was thrown during an assignment (i.e. `operator=`, `emplace`) into a `variant` object and is left in an indeterminate state.

```
bad_variant_assign(void* v_ptr) noexcept;
```

Effects: Constructs an object of class `bad_variant_assign`.

Postcondition: `variant_ptr() == v_ptr`.

```
bad_variant_assign(const bad_variant_assign&) noexcept;
bad_variant_assign& operator=(const bad_variant_assign&) noexcept;
```

Effects: Copies an object of class `bad_variant_assign`.

```
void* variant_ptr() const noexcept;
```

Returns: The pointer to the `variant` that is left in an indeterminate state.

Remarks: If multiple `variant` objects are left in the indeterminate state during the propagation of `bad_variant_assign`, the pointers to those variants will be captured within nested `bad_variant_assign` exceptions.

```
virtual const char* what() const noexcept;
```

Returns: An implementation-defined NTBS.

Remarks: The message may be a null-terminated multibyte string (17.5.2.1.4.2), suitable for conversion and display as a `wstring` (21.3, 22.4.1.4).

20.N.5 In-Place Construction [variant.in.place]

`in_place` is an overloaded function used to disambiguate the overloads of constructors and member functions of that take arguments (possibly a parameter pack) for in-place construction.

`variant` has constructors with

- `decltype(in_place<I>) == in_place_t (&)(integral_constant<size_t, I>)` as the first parameter which indicates that an object of type `tuple_element_t<I, variant<Types...>>` should be constructed in-place.
- `decltype(in_place<T>) == in_place_t (&)(T)` as the first parameter indicates that an object of type `T` should be constructed in-place.

20.N.6 Null Variant [variant.nullvar]

```
struct nullvar_t {};
```

`nullvar_t` is an empty class type used as an alternative of a `variant` which allows the user to opt-in for the presence of a conceptual representation of the empty state. [*Note*: The class template `variant` does not provide any special behavior for `nullvar_t`. The result of this is that `variant` behaves consistently with any custom representation of the empty state. — *end note*]

```
constexpr nullvar_t nullvar{};
```

`nullvar` is a constant of type `nullvar_t` that is used as an alternative of a `variant` which allows the user to opt-in for the presence of a conceptual representation of the empty state.

20.N.7 Element Access [variant.elem]

```
template <size_t I, class... Types>
constexpr remove_reference_t<tuple_element_t<I, tuple<Types...>>>* get(
    variant<Types...>* v) noexcept;
```

Effects: Equivalent to

```
    using T = tuple_element_t<I, tuple<Types...>>
    return const_cast<remove_reference_t<T>*>(
        get<I>(static_cast<const variant<Types...>*>(v)));
```

```
template <size_t I, class... Types>
constexpr remove_reference_t<const tuple_element_t<I, tuple<Types...>>>* get(
    const variant<Types...>* v) noexcept;
```

Let `J` be the constant expression of `v.index()`.

Requires: `I < sizeof...(Types)`

Returns: A pointer to the contained value of `v` if `v != nullptr && I == J` else `nullptr`

```
template <class T, class... Types>
constexpr remove_reference_t<T>* get(variant<Types...>* v) noexcept;

template <class T, class... Types>
constexpr remove_reference_t<const T>* get(const variant<Types...>* v) noexcept;
```

Let `I` be the index of `T` in `Types...`.

Requires: `T` is in `Types...` && `T` is unique within `Types...`.

Effects: Equivalent to `return get<I>(v)`

Returns: `get<I>(v)`

```
template <size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>& get(variant<Types...>& v);
```

Effects: Equivalent to

```
    using T = tuple_element_t<I, tuple<Types...>>;
    return const_cast<T&>(
        get<I>(static_cast<const variant<Types...>&>(v)));
```

```
template <size_t I, class... Types>
constexpr const tuple_element_t<I, tuple<Types...>>& get(
    const variant<Types...>& v);
```

Effects: Equivalent to

```
    auto *result = get<I>(&v);
    return result ? *result : throw bad_variant_access{};
```

```
template <size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>&& get(variant<Types...>&& v);
```

Effects: Equivalent to

```
    using T = tuple_element_t<I, tuple<Types...>>;
    return static_cast<T&&>(get<I>(v));
```

```
template <size_t I, class... Types>
constexpr const tuple_element_t<I, tuple<Types...>>&& get(
    const variant<Types...>&& v);
```

Effects: Equivalent to

```
    using T = tuple_element_t<I, tuple<Types...>>;
    return static_cast<const T&&>(get<I>(v));
```

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

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

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

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

Let `I` be the index of `T` in `Types...`.

Requires: `T` is in `Types...` && `T` is unique within `Types...`

Effects: Equivalent to `return get<I>(v);`

20.N.8 Relational Operators [variant.rel]

```
template <class... Types>
constexpr bool operator==(const variant<Types...>&lhs,
                          const variant<Types...>&rhs);
```

Requires: For all `I` where `I` $\in$ `[0, sizeof...(Types`$_I$`)]`, `get<I>(lhs) == get<I>(rhs)` is a valid expression returning a type that is convertible to `bool`.

Returns: `true` if `lhs.index() == rhs.index() && get<I>(lhs) == get<I>(rhs)` where `I` is the constant expression of `lhs.index()`, otherwise `false`.

```
template <class... Types>
constexpr bool operator<(const variant<Types...>&lhs,
                         const variant<Types...>&rhs);
```

Requires: For all `I` where `I` $\in$ `[ 0, sizeof...(Types`$_{\texttt{I}}$`)]`, `get<I>(lhs) < get<I>(rhs)` is a valid expression returning a type that is convertible to `bool`.

Returns: `true` if `std::tie(lhs.index(), get<I>(lhs)) < std::tie(rhs.index(), get<I>(rhs))` where `I` is the constant expression of `lhs.index()`, otherwise `false`.

20.N.9 Specialized Algorithms [variant.special]

```
template <class... Types>
void swap(variant<Types...>& lhs,
          variant<Types...>& rhs) noexcept(noexcept(lhs.swap(rhs)));
```

Effects: Equivalent to `lhs.swap(rhs)`.

20.N.10 Hash Support [variant.hash]

```
template <> struct hash<nullvar_t>;
```

The template specialization shall meet the requirement of class template `hash` (20.9.13).

```
template <class... Types> struct hash<variant<Types...>>;
```

Requires: The template specialization `hash<U>` shall meet the requirements of class template `hash` (20.9.13) for all `U` in `Types...`

The template specialization shall meet the requirement of class template `hash` (20.9.13).

20.N.11 Visitation [variant.visit]

```
template <class R = unspecified:deduce_tag, class... Variants>
unspecified:TypeSwitch<R, Variants&&...> type_switch(Variants&&... vs);
```

Effects: Constructs a callable object `TypeSwitch` which holds references to `forward<Variants>(vs)...`. Because the result may contain references to temporary variables, a program shall ensure that the return value of this function does not outlive any of its arguments. (e.g., the program should typically not store the result in a named variable).

```
template <class R, class... Variants>
struct TypeSwitch {
  explicit TypeSwitch(Variants...);

  template <class F>
  see below TypeSwitch::operator()(F&&) const;

  template <class... Fs>
  see below TypeSwitch::operator()(Fs&&...) const;
```

```
  template <template <class...> class F, class... Args>
  see below TypeSwitch::operator()(const typed_visitor<F, Args...>&) const;
};

explicit TypeSwitch(Variants... vs);
```

Effects: Holds references to `vs...`.

```
template <class F>
see below TypeSwitch::operator()(F&& f) const;
```

Let `vs...` be the references to instances of `Variants...` that *`TypeSwitch`* holds.

Let $\texttt{Variant}_\texttt{I}$ be the $\texttt{I}^{\text{th}}$ type in `decay_t<Variants>>...`.

Let $\texttt{Types}_\texttt{I}\texttt{...}$ be the template parameter pack `Types...` of $\texttt{Variant}_\texttt{I}$.

Requires: `invoke(forward<F>(f), get<`$\texttt{I}_0$`>(`$\texttt{v}_0$`), ..., get<`$\texttt{I}_N$`>(`$\texttt{v}_N$`)) OR forward<F>(f).template operator()<`$\texttt{I}_0$`, ..., `$\texttt{I}_N$`>(get<`$\texttt{I}_0$`>(`$\texttt{v}_0$`), ..., get<`$\texttt{I}_N$`>(`$\texttt{v}_N$`))` must be a valid expression for all $\texttt{I}_\texttt{I}$ where $\texttt{I}_\texttt{I} \in$ `[ 0, sizeof...(`$\texttt{Types}_\texttt{I}$`)]`.

Effects: Let `Is...` be the constant expression of `vs.index()...`. Equivalent to `forward<F>(f).template operator()<Is...>(get<Is>(vs)...)` if it is a valid expression, otherwise equivalent to `invoke(forward<F>(f), get<Is>(vs)...)`;

Remarks: If `!is_same_v<R, `*`unspecified`*`:deduce_tag>` is `true`, the return type of this function is `R`.

Otherwise, if `decay_t<F>` contains a member typedef `result_type`, the return type is `typename decay_t<F>::result_type`.

Otherwise, the return type is deduced by `decltype(auto)`.

```
template <class... Fs>
see below TypeSwitch::operator()(Fs&&... fs) const;
```

Effects: Equivalent to `(*this)(overload(forward<Fs>(fs)...)); // P0051`

5 Terminology

This section clearly defines some of the terminology used within this paper. There were some discrepancy around the understanding of terms such as `discriminated union`, `sum type`, `empty state`, and `null state`. It will be worthwhile to be accurately define them for clear and efficient communication.

5.1 Discriminated Union vs Sum Type

In set theory, a `discriminated union` (or disjoint union) of a family of sets $S_0, S_1, ..., S_N$ is a modified union operation that augments the elements with the index of the originating set. Formally,

this is defined as: $\sqcup S_i = \cup\{(x, i) : x \in S_i\}$. For example, $S_0 = \{1, 2, 3\}$ and $S_1 = \{1, 2\}$, $S_0 \sqcup S_1 = \{(1, 0), (2, 0), (3, 0), (1, 1), (2, 1)\}$.

In type theory, a sum type of a family of types $T_0, T_1, ..., T_N$ is a discriminated union of the types, except that augmentation is typically done with a name, rather than an index. In functional languages such as Haskell and OCaml, the name is represented as a type constructor. For example, the declaration of a sum type `Int + String` is required to be tagged with a name like so: `data Identifier = Id Int | Name String`, where `Id` is the tag of `Int` and `Name` is the tag of `String`. The notation for sum type is typically `T0 + T1` which derives from the sum relationship of the cardinality of the types: $|T0 + T1| = |T0| + |T1|$.

5.2 Empty State

A state in which the `variant` has none of its alternatives constructed.

5.3 Null State

A state in which the `variant` contains a value that represents the conceptual emptiness of a variant. All operations are valid. For example, the user must explicitly handle the null case for the visitation operation.

5.4 Valid, but Unspecified State

A state in which the `variant` has none of its alternatives constructed, but the discriminator of the `variant` is in a specified state. The valid operations in this state are: assignment (e.g. `operator=`, `emplace`), destruction, observation (e.g. `index`, `valid`), copy and move construction.

[*Example*: If an object `v` of type `std::variant<Types...>` is in a valid, but unspecified state, `v.index()` can be called unconditionally, but `get<I>(v)` can be called only if `v.index() >= 0` is `true`. — *end example*]

5.5 Indeterminate State

A state in which the `variant` has none of its alternatives constructed, and the discriminator of the `variant` is also in an unspecified state. The only valid operations in this state are: assignment (e.g. `operator=`, `emplace`), and destruction. Any other operations performed in this state have undefined behavior.

6 Design Decisions

6.1 Empty State

As defined in 5 Terminology, an empty state of a `variant` is a state where none of the alternatives are constructed. The following is an example that illustrates the problem:

```
std::variant<T, U> v(T(/* ... */));
U u;
v = u;
```

When `v` is assigned with `u`, it must destroy the contained value of type `T` and in-place construct a value of type `U`. If an exception is thrown during the construction `U`, `v` would be in an empty state, since it has already destroyed the initial value of type `T`.

The following subsections describe the various strategies considered to handle the exceptional case which can arise when some alternatives have move constructors that can throw. Note that out of all the design discussions of a standardized `variant`, this is the most contentious issue. Each approach has at least one disadvantage which is claimed absolutely unacceptable to a group of people. The goal of this paper is to choose a design that minimize the compromises, and also to leave enough room for extensibility for those who absolutely do not agree. The strategy proposed in this paper is mentioned last, highlighted in blue.

6.1.1 Additional Requirements on Alternatives

The general theme of this subsection is to enforce additional requirements on the alternatives. There are various additional requirements which can be enforced to varying degrees.

For example, we could add additional requirements such as (1) all alternatives must be nothrow move constructible, or (2) all alternatives must be nothrow default constructible and nothrow swappable.

If the additional requirements are not satisfied, we could (1) disallow `variant` as a whole, or (2) delete the assignment operations.

Exception Safety: Strong

Pros: New types should have nothrow move constructors, and provides strong exception safety guarantee.

Cons: Regardless of whether most new types should be nothrow move constructible or not, the standard library is (and should continue to be) resilient to the existence of types with move constructors that can throw. The standard library currently actively supports types with move constructors that can throw (e.g. `std::vector::push_back`) by use of utility functions such as `std::move_if_noexcept`.

Furthermore, the conditions in which a type can throw during its move construction is not as rare as we would like. The following are conditions in which a move can throw:

- Types in the standard library with move constructors that can throw (e.g. `std::set`),

- Legacy types that do not have a move constructor would fall-back to copy construction which can throw. This may be a type that the user can add a `noexcept` move constructor to, but it may also be an external library type that the user has no control over.
- All `const`-qualified types. A type with `noexcept` move constructor (e.g. `std::string`) can throw during a move construction if it is `const`-qualified (e.g. `const std::string`).

6.1.2 Double Storage

The general theme of this subsection is to require double storage in situations where additional requirements are not satisfied by the alternatives. It could be applied in varying degrees within the spectrum of "minimize the cases in which double storage is introduced" to "always use double storage".

Currently, the narrowest condition known to require double storage is if at least `two` alternatives can throw during move construction.

Exception Safety: Strong

Pros: Under conditions in which double storage is required, people should accept their fate in having to pay for the performance penalties.

Also, this is an issue that can be deferred as an implementation detail which will continue to improve over time, and such implementation detail should not hinder the interface.

Cons: While the double storage strategy could be seen as an implementation detail that should not affect the interface, I would argue that `std::vector`'s contiguous memory guarantee could also be seen as an implementation detail that should not have affected the interface. Consider the exception safety specification of `std::vector::push_back`:

> If the `T`'s move constructor is not `noexcept` and `T` is not `CopyInsertable` into `*this`, `std::vector` will use the throwing move constructor. If it throws, the strong exception safety guarantee is waived and the effects are unspecified.

If `std::vector` was designed without the contiguous memory guarantee which could have been seen as an "implementation detail", it would have ended up similar to `std::deque` where `push_back` would always have strong exception safety guarantee.

The point is that giving up single storage in some cases for `std::variant` would be analogous to giving up contiguous memory in some cases for `std::vector`.

6.1.3 Null State as a Required Alternative

As defined in 5 Terminology, a null state is a state in which `variant` contains a value that represents the conceptual emptiness of a `variant`. This approach is to make the null state a required state of a `variant`. This requirement could be enforced implicitly or explicitly:

- **Implicit**: `nullvar_t` is always implicitly injected as an alternative. That is, `variant<int, std::string>` holds one of `int`, `std::string`, or `nullvar_t`

- **Explicit**: `nullvar_t` is required to always explicitly be provided as an alternative. That is, `variant<int, std::string>` is not allowed. Must always provide `nullvar_t`.

This approach is analogous to a pointer, and therefore inherits all of the characteristics of a pointer. The following are arguments for and against regarding the inherited characteristics.

Exception Safety: Basic

Pros: This approach is simple, and analogous to a pointer which is a well understood primitive of the language, and people know how to deal with them.

Cons: It requires the user of a `variant` to account for the null state everywhere. This is a major disadvantage of a pointer that have been referred to by Tony Hoare as [The Billion Dollar Mistake]. References were introduced as a solution to provide non-null semantics for a pointer. While it is possible to provide a separate class template which is analogous to a reference, it would be preferred to introduce non-null semantics by default. Since opting in for the null state given non-null semantics is trivial, while the converse is not.

6.1.4 Valid, but Unspecified State

A state in which the `variant` has none of its alternatives constructed, but the discriminator of the `variant` is in a specified state. This state was introduced at WG21's 2015 Lenexa meeting, and is the approach taken by [N4542].

This state is visible via the `bool`-returning function `valid()`. The function is named `valid()` because even though the `variant` is technically in a valid state, for all intents and purposes it is invalid. It may be worth considering alternative names in order to avoid confusion (e.g. `operator bool`, `unspecified`, `usable`, `ready`).

Exception Safety: Basic

Pros: It handles all types including ones that can throw during move construction, and it does not incur any performance overhead. The null state is opt-in, so there is no null state by default.

Cons: Since this state is visible and partially well-defined, it's perfectly valid for users to call the `valid()` function anywhere in the code to test for the valid, but unspecified state. This is contrary to its intended use, which is to check for the validity of the `variant` within a `catch` clause. The following are somewhat contrived examples of intended and unintended use cases.

```
// intended usage:
void F(variant<T, U>& v, const variant<T, U>& w) {
  try {
    v = w;
  } catch (const std::exception& ex) {
    if (!v.valid()) {
      // Failed during in-place construction, old value is unknown.
    } else {
      // Failed during temporary construction, old value is in ‘v‘.
    }
  }
}
```

```
// intended usage:
void G(variant<T, U>& v, const variant<T, U>& w) {
  // Well, even if I was to catch the exception, I can't do anything about it.
  // So... just propagate it!
  v = w;
}

void H(variant<T, U>& v, const variant<T, U>& w) {
  try {
    G(v, w);
  } catch (const std::exception& ex) {
    if (!v.valid()) {
      // Failed during in-place construction, old value is unknown.
    } else {
      // Failed during temporary construction, old value is in `v`.
    }
  }
}
```

```
// unintended, valid usage:
void F(variant<T, U> v) {
  assert(v.valid());
  // Ok, use v.
}
```

```
// unintended, valid usage:
void F(const variant<T, U>& v) {
  assert(v.valid());
  // Ok, use v.
}
```

```
// unintended, valid usage:
void F(variant<T, U>& v) {
  if (!v.valid()) {
    // Make v valid.
    v = T{};
  }
  // Ok, use v.
}
```

Ideally, we should eliminate the unintended usage of the `valid()` function.

6.1.5 Indeterminate State

A state in which the `variant` has none of its alternatives constructed, and the discriminator of the `variant` is also in an unspecified state. The only valid operations in this state are: assignment (e.g. `operator=`, `emplace`), and destruction. Any other operations performed in this state have undefined behavior.

Exception Safety: None

Pros: It handles all types including ones that can throw during move construction, and it does not incur any performance overhead. The null state is opt-in, so there is no null state by default.

Unlike the valid, but unspecified state, this state is unobservable, and any operations aside from assignment and destruction trigger undefined behavior. With these properties, a function can and is required to make stronger assumptions about the state of a `variant`.

```
void F(variant<T, U> v) {
  // v must be valid.
}
```

```
void F(const variant<T, U>& v) {
  // Assume v is valid.
}
```

```
void F(variant<T, U>& v) {
  // Either:
  //   - Assume v is valid, read and/or mutate the value, OR
  //   - Assume v is in an indeterminate state and assign a value to it.
}
```

Note that the above conditions are exactly what we would and be required to assume if we were to replace `variant<T, U>` with `int`.

Cons: Since all other operations aside from assignment and destruction trigger undefined behavior, this state is more dangerous than the valid, but unspecified state. This also means that under the conditions in which an exception is thrown during `variant` assignment, we provide no exception safety guarantees.

6.2 Default Construction

Many have claimed desirable for a `variant` to be default constructible for ease of use in containers such as `std::vector`, `std::map`, and `std::array`. The requirements of a default constructor for `std::vector` and `std::map` are not as strong, as there are alternative constructors and functions available to better support types without a default constructor. For example, `std::vector` provides `std::vector::vector(size_t count, const T&)` which can be used instead of `std::vector::vector(size_t count)`, and `std::map` provides `std::map::insert` which can be used instead of `std::map::operator[]`. While there are alternative approaches, it is indeed more difficult to use, as most users would prefer `m[key] = value;` over `m.insert(std::make_pair(key, value));`

For `std::array`, there are no good alternatives. One would usually resort to using `std::vector` instead which may be undesirable due to the incurred dynamic allocation and the loss of fixed size guarantee.

This section discusses various default constructed states. The default constructed state proposed in this paper is mentioned last, highlighted in blue.

6.2.1 First Type

This is the design that LEWG opted for at the WG21's 2015 Lenexa meeting. This is the behavior of a `union` when value initialized, and it makes sense for a `variant` if the assumption is that it should be modeled after `union`.

Pros: This is what makes the most sense if we **had to** default construct one of the types, and this state initializes to a fully valid state.

Cons:

- `variant` remains non-default-constructible if the first type is non-default-constructible.
- The exception-safety guarantee depends on the first type.
- The runtime complexity guarantee depends on the first type.
- The behavior of default construction of `variant` changes based on the *order* of types.

Some have argued that the varying behavior of default construction across different instantiations of `variant` should not be surprising since `variant<T, U>` and `variant<U, T>` are separate types as far as the language is concerned. However, consider that `vector<bool>` and `vector<int>` are also separate types as far as the language is concerned and the amount of pain that comes from the inconsistent interfaces and behavior. Even though they are indeed separate types, there is still an intuitive expectation that such a class template has consistent behavior across instantiations with different types.

6.2.2 Valid, but Unspecified State

[N4542] mentions that LEWG has opted against default constructing into a valid, but unspecified state in order to limit the paths of getting into such a state to the exceptional cases.

Pros: Under the assumption that default construction of `variant` is desirable, it would be desirable for **any** `variant` to be default constructible. Default constructing into the valid, but unspecified state is one option that allows any `variant` to be default constructible regardless of the types it contains.

By default constructing into this state, the behavior of default construction stays consistent for any `variant`, provides the `noexcept` guarantee, and constant runtime complexity guarantee.

Cons: As discussed in 5.4 Valid, but Unspecified State, recall that a valid, but unspecified state is visible. As such, default constructing into this state would make it a common state that users can and would be required to check for outside of the `catch` clause. That is, the following code becomes perfectly valid:

```
void F(variant<T, U>& v) {
  if (!v.valid()) {
    // Make v valid.
    v = T{};
  }
  // Ok, use v.
}
```

```

int main() {
  variant<T, U> v;
  F(v);
}
```

In order to avoid such code, the user would have to add the "`v.valid()` is `true`" precondition to function `F`. Naturally, this precondition would exist for all operations of `variant` except assignment (e.g. `operator=`, `emplace`), destruction, observation (e.g. `index`, `valid`), and copy and move construction. This design would only be slightly better than requiring the null state to always exist, or deferring to using `std::optional<variant<Types...>>`.

6.2.3 Indeterminate State

Pros: Refer to 6.1.5 Indeterminate State and 6.2.2 Valid, but Unspecified State for previous discussions regarding the advantages of having this kind of state.

Unlike the valid, but unspecified state, the only valid operations of this state are assignment (e.g. `operator=`, `emplace`), and destruction. As mentioned in 6.1.5 Indeterminate State, this means that functions can make stronger assumptions about the state of a `variant` object.

Note that this behavior is similar how `int` behaves:

```
int x;  // uninitialized
x = 42;  // assign
```

```
std::variant<int, std::string> v;  // indeterminate
v = 42;  // assign
```

The argument is that validity checks of a `variant` are no longer necessary nor possible, the same way validity checks of an `int` are not necessary nor possible. It is required that a `variant` object in an indeterminate state be assigned a value before use, the same way it is required that uninitialized `int` be assigned a value before use, in order to avoid undefined behavior.

This is true for `enum` as well, which is an existing discriminated union with value semantics.

```
enum class Color { Blue, Green, Red };
Color color;  // uninitialized
color = Color::Blue;  // assigned
```

Another argument is one of efficiency. Given a size `N`, the following are different ways to initialize a `std::array<int, N>`, and `std::vector<int>` of size `N`:

```
std::array<int, N> v;  // uninitialized

std::array<int, N> v = {};  // initialized to 0
```

```
// avoid initialization of N integers
std::vector<int> v;
v.reserve(N);
```

```cpp

// initialized to 0
std::vector<int> v(N);
```

It's common practice to avoid the cost of initialization for performance gains. The argument here is that if there are performance gains from not initializing `int` to `0`, there must also be performance gains from not default constructing the first type of `variant`.

This design has the following properties:

- Any `variant<Types...>` is default constructible.
- Consistent behavior of default construction for all `variant<Types...>`.
- `noexcept` exception-safety guarantee.
- Constant runtime complexity guarantee.

Cons: While the behavior of this state is **similar** to `int`, it's not **equivalent** due to the fact that classes cannot differentiate between default initialization and value initialization.

```cpp
int x;  // uninitialized
int y{};  // initialized to 0
```

```cpp
std::variant<int, std::string> v;  // indeterminate
std::variant<int, std::string> w{};  // indeterminate
```

It should however be noted that this caveat already exist for atomic types in the `<atomic>` library.

```cpp
std::atomic<int> x;  // uninitialized
std::atomic<int> y{};  // uninitialized
```

Lastly, it goes against one of the guidelines given in the [C++ Core Guidelines]: **ES.20: Always initialize an object** which recommends to avoid uninitialized state.

6.3 Null State

This section discusses the approaches in which the null state could be handled for `variant`. The strategy proposed in this paper is mentioned first, highlighted in blue.

6.3.1 `std::nullvar_t`

This approach is to simply introduce an empty class type `std::nullvar_t` analogous to `std::nullopt_t`. Users would opt-in for a null state by specifying `std::nullvar_t` as one of the alternatives.

Furthermore, `variant` does not provide special behavior for `std::nullvar_t`. The main motivation is to keep consistent behavior of `variant<Types...>`. As mentioned in 6.2.1 First Type, there is an expectation that a class template has consistent behavior across instantiations with different types. This follows that design principle of not introducing special behavior due to involvement of a type that we consider special. This, in turn, facilitates generic programming.

- The types that need to be handled for visitation all appear in the type list of a `variant`.
- Consistent relationship between
  - (`std::nullvar_t`, `std::variant`)
  - (`std::nullopt_t`, `std::optional`)
  - (`std::nullptr_t`, `T*`)
- Since there is no special behavior provided for `std::nullvar_t`, users are free to provide a custom type that represents the null state within their project. (e.g. `empty_t` in Adobe ASL, `none_t` in Boost, `None` in Facebook Folly, `Nothing` in Apache Mesos)

Consider the symmetry in the handlers of a generalized `type_switch` where types such as pointers and `std::optional` are supported:

```
std::optional<std::size_t> x;
std::variant<int, std::string, std::nullvar_t> y;
int *z;
/* ... */
std::type_switch (x, y, z) (
  [](std::size_t   , int                , int           ) { /* ... */ },
  [](std::size_t   , int                , std::nullptr_t) { /* ... */ },
  [](std::size_t   , const std::string& , int           ) { /* ... */ },
  [](std::size_t   , const std::string& , std::nullptr_t) { /* ... */ },
  [](std::size_t   , std::nullvar_t     , int           ) { /* ... */ },
  [](std::size_t   , std::nullvar_t     , std::nullptr_t) { /* ... */ },
  [](std::nullopt_t, int                , int           ) { /* ... */ },
  [](std::nullopt_t, int                , std::nullptr_t) { /* ... */ },
  [](std::nullopt_t, const std::string& , int           ) { /* ... */ },
  [](std::nullopt_t, const std::string& , std::nullptr_t) { /* ... */ },
  [](std::nullopt_t, std::nullvar_t     , int           ) { /* ... */ },
  [](std::nullopt_t, std::nullvar_t     , std::nullptr_t) { /* ... */ }
);
```

6.3.2 `std::optional<std::variant<Types...>>`

This is an intuitive approach since `std::optional` exists to provide the notion of a null state. Wrapping the `variant` inside of an `optional` would augment the null state to a list of types `Types...` as desired. However, this introduces an extra level of indirection that needs to be observed and dereferenced. This matters because generally flat control flow is more readable and easier to reason about compared to nested control flow. Consider the following control flow with `std::optional<std::variant<int, std::string>>`:

```
std::optional<std::variant<int, std::string>> x, y;
/* ... */
if (x && y) {
  std::type_switch (*x, *y) (
    [](int, int) { /* handle int, int. */ },
    [](int, const std::string&) { /* handle int, string. */ },
    [](const std::string&, int) { /* handle string, int. */ },
    [](const std::string&, const std::string&) { /* handle string, string. */ }
```

```
  );
} else if (x) {
  std::type_switch (x) (
    [](int) { /* handle int, null. */ },
    [](const std::string&) { /* handle string, null. */ },
  );
} else if (y) {
  std::type_switch (y) (
    [](int) { /* handle null, int. */ },
    [](const std::string&) { /* handle null, string. */ },
  );
} else {
  /* handle null, null */
}
```

Compared to the control flow with `std::variant<int, std::string, std::nullvar_t>`:

```
std::variant<int, std::string, std::nullvar_t> x, y;
/* ... */
std::type_switch (x, y) (
  // no need for comments, the signature tells you exactly what types are being handled.
  [](int               , int               ) { /* ... */ },
  [](int               , const std::string&) { /* ... */ },
  [](int               , std::nullvar_t    ) { /* ... */ },
  [](const std::string&, int               ) { /* ... */ },
  [](const std::string&, const std::string&) { /* ... */ },
  [](const std::string&, std::nullvar_t    ) { /* ... */ },
  [](std::nullvar_t    , int               ) { /* ... */ },
  [](std::nullvar_t    , const std::string&) { /* ... */ },
  [](std::nullvar_t    , std::nullvar_t    ) { /* ... */ }
);
```

We can see that `std::variant<int, std::string, std::nullvar_t>` enables a flatter control flow compared to `std::optional<std::variant<int, std::string>>`.

6.3.3 Separate Class Template: `std::nullable_variant`

The idea is to introduce a separate class template to represent a nullable `variant`, similar to the way pointer and reference are distinct type modifiers. However, given that a `variant` is a discriminated union, it seems more natural that this case be generically handled where a null state is simply one of the states of the discriminated union.

The main advantage of introducing this class template would be that an empty state of any kind (e.g. valid, but unspecified, indeterminate) need not exist.

There are at least 2 possible interfaces:

1. Essentially an alias for `std::optional<std::variant<Types...>>`
2. Modified behavior based on `std::variant<std::nullvar_t, Types...>`

The disadvantages of (1) are discussed in 6.3.2 `std::optional<std::variant<Types...>>`.

For (2) the main disadvantages are:

- `std::nullvar_t` becomes special which disallows the use of a custom null type.
- Either `std::nullvar_t` is disallowed for `variant`, or users need to be aware of the subtle differences between `variant<std::nullvar_t, Types...>` vs `nullable_variant<Types...>`.
- In generic programming, it would be required to handle `variant` and `nullable_variant` separately even in cases where it can be handled generically.

6.4 Discriminator

This section discusses the possible discriminator to refer to an alternative of a `variant`. The strategy proposed in this paper is mentioned last, highlighted in blue.

6.4.1 Name

When identifiers are used to distinguish the members of a discriminated union, it is a sum type. This approach is used by `union` for example, where each member of a `union` has an associated identifier. Given `union { T t; U u; };`, `x` and `y` are distinct because the identifier is the discriminator. Since there are no identifiers involved in `variant<T, U>`, in order to support this feature, `variant` would need to be introduced at the language-level rather than at the library-level.

6.4.2 Type

This approach uses the type of the alternatives as the discriminator of a discriminated union.

Pros: Since arbitrary new types can be created with unique names, this enforces that every alternative be uniquely identified by a type which maintains the reference relationship even in the event of reordering. For example, given `variant<Circle, Square, Triangle> shape; get<Circle>(shape);` retrieves `Circle` even if we change the declaration of the shape variable to `variant<Square, Triangle, Circle> shape;`

Furthermore, the inability to represent multiple occurrences of a single type `T` as distinct states can be addressed by introducing wrapper classes `X` and `Y` where each of them wrap an instance of `T`. This is useful also because it introduces a name that is distinctive at the type-level (unlike type alias declarations).

Lastly, The `variant` interface becomes simpler since there is no need to support index-based operations such as `variant(in_place<0>, 42)` and `get<0>(v)`.

Cons: The notion of a named discriminator and the corresponding type are conflated into a single type. It also leads to the question: should multiple occurrences of a single type `T` in `Types...` (e.g. `variant<T, T>`) be disallowed, or allowed, but do not represent distinct states?

Neither of these definitions are accurate in terms of a discriminated union from set theory nor sum type from type theory. Therefore, this approach is more suitable to be considered a workaround technique to emulate a sum type, rather than being part of library design.

6.4.3 Index

This approach uses the index of an alternative as the discriminator. This behavior of implicitly assigned indices is consistent with `std::tuple`, where the elements of `std::tuple` can be referred to with its their indices, or by their type if the type is unique within `Types...`.

Pros: This definition is accurate to the definition of discriminated union, and the interface is consistent with `std::tuple`. The workaround technique of introducing new types and using the type-based operations are still applicable.

Cons: While using the indices may seem essentially equivalent to using associated names, a significant difference is that the *order* of types become sensitive in the index-based operations. Suppose the type of `v` in the expression `get<0>(v);` is `variant<T, U>`. If the type is later changed to `variant<U, T>`, or `variant<V, T, U>`, a different member is retrieved.

This is contrary to using the name as the discriminator. Consider `union` as the example of name being used as the discriminator (except union does not keep track of the discriminator itself). Regardless of whether the type of `u` in the expression `u.x` is `union { X x; Y y; };` or `union { Y y; X x; };`, the member of type `X` is retrieved.

This is an inherent difference in the meaning of the expressions `get<0>(v)` and `u.x`, the former says "get me whatever is the first alternative" whereas the latter says "get me the alternative that is tagged with `x`".

6.5 Visitation: Interface

This section describes a possible visitation mechanism and does not preclude other mechanisms. For example, [P0050] is a draft of a pattern matching proposal at the library-level, while [Mach7] is an experimental library solution to [N3449] which aims to bring pattern matching at the language-level. The mechanism being proposed here is a type switch which is much simpler than generalized pattern matching, and only specifies the aspects involving `variant`.

6.5.1 `visit(F, Variants...);`

[N3915] proposed `apply` which takes a function object `F` and a tuple `Tuple`, and dispatches `F` with the elements of `Tuple`. This function is similar, but for `variant`. That is, `visit` takes a function object `F` and `Variants...`, and dispatches `F` with the content of each the `Variants...`.

Although the similarity to `apply` seem desirable for consistency, I believe there are major drawbacks at the callsite. The inherent difference between `apply` and `visit` is that `apply` only requires a single handler that handles the elements of the `tuple`, whereas `visit` (almost always) requires many handlers that handle the cartesian product of the alternatives of the `variant` objects.

Consider a few sample usages of visiting a `variant<int, std::string>` The following are 3 of the possible ways to provide the handlers via the `visit` interface.

- Function object which requires out-of-line definition.
- Generic lambda to provide a single generic handler inline.

- `overload` to provide specific handlers inline. // P0051

```cpp
int main() {
  std::variant<int, std::string> v(42);

  // Function object which requires out-of-line definition.
  struct Print {
    void operator()(int value) const { std::cout << "int: " << value; }
    void operator()(const std::string& value) const { std::cout << "string: " << value; }
  };
  std::visit(Print{}, v);

  // Generic lambda to provide a generic handler inline.
  std::visit([](const auto& value) { std::cout << value; }, v);

  // ‘overload‘ to provide specific handlers inline.  // P0051
  std::visit(
    overload([](int value) { std::cout << "int: " << value; }
             [](const std::string& value) { std::cout << "string: " << value; }),
    v);
}
```

6.5.2 `type_switch(Variants...)(Fs...);`

The type switch mechanism here is similar to `visit`, but there are a couple of differences.

- The `variant` objects come before the handlers.
- `overload` is no longer necessary.

```cpp
int main() {
  std::variant<int, std::string> v(42);

  // Function objects can still be defined out-of-line.
  struct Print {
    void operator()(int value) const { std::cout << "int: " << value; }
    void operator()(const std::string& value) const { std::cout << "string: " << value; }
  };
  std::type_switch (v) ( Print{} );

  // Generic lambda to provide a generic handler inline.
  std::type_switch (v) ( [](const auto& value) { std::cout << value; } );

  // No need for ‘overload‘.
  std::type_switch (v) (
    [](int value) { std::cout << "int: " << value; },
    [](const std::string& value) { std::cout << "string: " << value; }
  );
}
```

This approach is favored over the `visit` function considering the inherent difference that `variant` visitation require multiple handlers in overwhelming number of cases. It is also more reminiscent of existing language constructs such as the `switch` statement in C++ and various forms of `match`

statements in functional languages such as OCaml. The name `type_switch` was chosen over `match` to

- leave `match` available for language-level pattern matching
- avoid potential confusion with the `match`-related names in the `<regex>` library.

6.6 Visitation: Return Type

This section explores various options regarding the return type of visitation. Should the return type be deduced to something sensible? or should it be required that all of the handlers return the same type?

Consider the following example:

```
std::variant<int, std::string> v(42);
auto result = std::type_switch (v) (
  [](int value) /* -> int */ { return value; },
  [](const std::string& value) /* -> std::size_t */ { return value.size(); }
);
// What is the result of decltype(result)? Does this even compile?
```

6.6.1 `std::common_type_t<return_types...>`

The approach here is to deduce the final return type as the common type of return types of each of the handlers. While this approach may seem reasonable, it is quite dangerous as we can very easily have loss of data and/or precision. For example, the deduced return type of the example above would be `std::size_t` which is likely to be a silent error if the `int` handler returns a negative value.

6.6.2 `std::variant<return_types...>`

In order to prevent data loss and type information, we could try to deduce the return type to be `std::variant<return_types...>`. In the case of the above example, the return type would be `std::variant<int, std::size_t>`.

The immediately obvious disadvantage of this approach is that even if all return types are `T`, the final return type is `variant<T>` if duplicates removed, or `variant<T, T, ..., T>` otherwise, rather than `T`. While this can be mitigated by adding it as a special rule, it immediately starts to add complexity to the return type deduction rule for a `variant`.

Another case to keep in mind is the presence of `void`-returning functions. Consider a function object where the return types of `operator()` are: `int, std::string, void`. Since `void` does not add additional state to a variant, naively deducing the return type to be `variant<int, std::string, void>` is not sufficient. The actual deduced return type needs to be `variant<int, std::string, std::nullvar_t>`.

Other situations that may or may not require special rules:

- Two handlers with return types: `const X&` and `const Y&`. Should the deduced return type be `variant<const X&, const Y&>`, or `const variant<X, Y>&`?
- Two handlers with return types: `const D1*` and `const D2*`, and `D1` and `D2` both inherit from `B`. Should the return type be `variant<const D1*, const D2*>`, or should the covariant return type rule kick in and therefore result in `const B*`?

Return type deduction rules in C++ are already quite complex at the language-level today. Introducing this level of complexity at the library-level would be very much undesirable.

6.6.3 Same Type

Having considered the disadvantages of the two approaches above, this paper proposes that the return types of all of the handlers must have the same return type. This rule enforces the programmer to be explicit, does not suffer from loss of data/type, and is simple to remember.

In order to mitigate the need to repeatedly specify the final return type in each handler, an explicit return type can be specified as a template argument to `type_switch`.

```
std::variant<int, std::string> v(42);
auto result = std::type_switch<variant<int, std::size_t>>(v) (
  [](int value) { return value; },
  [](const std::string& value) { return value.size(); }
);
// decltype(result) == variant<int, std::size_t>
```

6.7 Visitation: Dispatch

This section describes how dispatching works for visitation of `variant` objects. The visitation mechanism proposed here requires exhaustive match, but again, it does not preclude other proposals to override or augment this proposal.

The handler can have a superset of all possible cases, the only requirement is that it covers all potential cases of the match. For example, consider the following code:

```
class Circle;
class Square;
class Rectangle;
class Rhombus;
class RightTriangle;
class EquilateralTriangle;

using Shape = variant<Circle, Square, Rectangle, Rhombus, RightTriangle, AcuteTriangle>;
using Quadrilateral = variant<Square, Rectangle, Rhombus>;
using Triangle = variant<RightTriangle, AcuteTriangle>;

// ‘get_area‘ can handle all of ‘Shape‘, ‘Quadrilateral‘, and ‘Triangle‘.
template <class... Types>
double get_area(const std::variant<Types...>& shape) {
  return type_switch (shape) (
    [](const Circle& that) { /* ... */ },
    [](const Square& that) { /* ... */ },
```

```
    [](const Rectangle& that) { /* ... */ },
    [](const Rhombus& that) { /* ... */ },
    [](const RightTriangle& that) { /* ... */ },
    [](const EquilateralTriangle& that) { /* ... */ }
  );
}
```

6.7.1 Unique Alternatives

When the alternatives consist of unique types, we simply provide handlers for each alternative.

6.7.2 Duplicate Alternatives

When there are duplicate alternative types, it gets a bit more complicated.

The following code is allowed:

```
variant<int, int, std::string> v;
/* ... */
type_switch (v) (
  [](int) { /* handles both int alternatives. */ },
  [](const std::string&) { /* handles string. */ }
);
```

This may seem crazy at first, but consider a common use case of the default handler that matches anything with `auto&&`, as well as the use of constrained function templates with SFINAE or Concepts to match a family of types.

```
variant<int, std::size_t, double, std::string> v;

/* ... */

// Matching multiple types with a single default handler:
type_switch (v) (
  [](int) { /* handle int. */
  [](auto&&) { /* handles int, std::size_t and string. */ },
);

// Matching subset of types with constrained function templates.
struct handler {
  template <typename T>
  std::enable_if<std::is_integral_v<T>> operator()(T) const {
    /* handles int, and std::size_t */
  }

  void operator()(double) const { /* handle double */ }
  void operator()(const std::string&) const { /* handle string */ }
};

type_switch (v) ( handler{} );
```

The expectation is that a handler is expected to generically handle whatever type it matches, and this is consistent even in the presence of duplicate types.

In the rare cases where it is useful to know the original type in which the value came from, an index-aware handler can be provided to retrieve the indices of the originating types. This situation can arise from duplicate types, but also from the consequence of allowing references.

```
int x = 42;
variant<int, int&> v(in_place<1>, x);
type_switch (v) ( [](int&) { /* matches both int, and int& */ );

// index-aware handler
struct index_aware {
  template <size_t I>
  void operator()(int&) const {
    /* still matches both int, and int&, but 'I' tells you which one. */
  }
};

type_switch (v) ( index_aware{} );
```

Since non-deduced template parameters cannot be specified in a lambda, the implication here is that an index aware handler must be defined as an out-of-line function object. However, considering that this is likely a rare use case, it is unlikely to be a big deal.

6.8 Miscellaneous

6.8.1 `constexpr`

This proposal stays consistent with `std::optional` in terms of its `constexpr`-ness. More sophisticated conditions such as `std::is_trivially_copy_constructible_v` to increase the `constexpr`-ness is not considered in this proposal.

6.8.2 `void`

`void` is allowed, and adds no state. This result follows from the fact that the proposed `variant` design accurately models the formal definition of a discriminated union. Since the `void` type has an empty set of values (i.e. 0 states), it has a + 0 effect on the number of states of a `variant` object.

While this is a theoretical argument, the practical argument is that it is not possible to use `void` as a match type for multi-visitation. Consider a single visitation scenario where `void` is used to indicate the null state.

```
variant<int, void> v;
/* ... */
type_switch (v) (
  [](int ) { /* ... */ },
  [](void) { /* ... */ }  // == []() { /* ... */ }
);
```

Now imagine generalizing that to multi-visitation:

```
variant<int, void> v, w;
/* ... */
type_switch (v, w) (
  [](int , int)  { /* ... */ },
  [](int , void) { /* ... */ },  // invalid
  [](void, int)  { /* ... */ },  // invalid
  [](void, void) { /* ... */ }   // invalid
);
```

Therefore even for practical reasons, there is clear benefit to introducing a `std::nullvar_t`.

6.8.3 References

References are allowed as an alternative.

7 Extensibility

This section exists to demonstrate how this `variant` design can be extended to support various features desired by polarizing groups. It is intentionally separated out from 6 Design Decisions, since extensibility in itself was not a driving factor in the proposed design. However, it is evident that starting from the proposed design, those who want different semantics can extend it to get the semantics they want, whereas trying to arrive at this design based on other alternatives would be much more difficult.

7.1 Disallow Default Construction

Given a `variant` that deletes the default constructor, it would be very difficult to extend that `variant` and make it default constructible. However, the converse is trivial.

```
template <class... Types>
class Variant : public std::variant<Types...> {
public:
  Variant() = delete;
};
```

7.2 Default Construct the First Type

Given a `variant` that default constructs to the first type, it would be very difficult to extend that `variant` and keep it uninitialized. However, the converse is trivial.

```
template <class Head, class... Tail>
class Variant : public std::variant<Head, Tail...> {
public:
  Variant() : std::variant<Head, Tail...>(Head{}) {}
};
```

7.3 Turn off Assignment

Given a `variant` that has assignment turned off, users who want assignment would need to implement their own. However, the converse scenario of turning off given functionality is very easy.

```
template <class... Types>
class NonAssignableVariant : public std::variant<Types...> {
public:
  template <std::size_t I, class... Args> void emplace(Args&&...) = delete;

  template <std::size_t I, class U, class... Args>
  void emplace(std::initializer_list<U>, Args&&...) = delete;

  template <class T, class... Args> void emplace(Args&&...) = delete;

  template <class T, class U, class... Args>
  void emplace(std::initializer_list<U>, Args&&...) = delete;

  template <class U> variant& operator=(U&&) = delete;

  variant& operator=(const variant&) = delete;
  variant& operator=(variant&&) = delete;
};
```

7.4 Turn off Assignment under Certain Conditions

Given a `variant` where assignment is turned off in some cases, users who want assignment even in those cases would need to override the behavior with their own implementation of assignment. Leveraging 7.3 Turn off Assignment, the converse scenario remains very simple.

```
template <class... Types>
class Variant
    : public std::conditional_t<
          (std::is_nothrow_move_constructible_v<Types> && ...),
          std::variant<Types...>,
          NonAssignableVariant<Types...>> {};
```

7.5 Nullable Variant

Given a `variant` that always carries a null state and default constructs into it as well as recovering to that state when an exception occurs during the move construction of assignment, it would be difficult to make a non-nullable variant out of it without adding runtime checks. Admittedly, augmenting the null state to a non-null `variant` and providing augmented behavior was not as trivial as I thought it would be, but it's still doable.

```
template <class... Types>
class Variant : public std::variant<std::nullvar_t, Types...> {
public:
  using super = std::variant<std::nullvar_t, Types...>;

```

```
  Variant() : super(std::nullvar) {}

  template <std::size_t I, class... Args>
  void emplace(Args&&... args) {
    try_([&] { super::template emplace<I>(std::forward<Args>(args)...); });
  }

  template <size_t I, class U, class... Args>
  auto emplace(std::initializer_list<U> init, Args&&... args)
      -> decltype(super::template emplace<I>(init, std::forward<Args>(args)...)) {
    try_([&] { super::template emplace<I>(init, std::forward<Args>(args)...); });
  }

  template <class T, class... Args>
  void emplace(Args&&... args) {
    try_([&] { super::template emplace<T>(std::forward<Args>(args)...); });
  }

  template <class T, class U, class... Args>
  auto emplace(std::initializer_list<U> init, Args&&... args)
      -> decltype(super::template emplace<T>(init, std::forward<Args>(args)...)) {
    try_([&] { super::template emplace<T>(init, std::forward<Args>(args)...); });
  }

  template <class U>
  auto operator=(U&& u) -> decltype(super::operator=(std::forward<U>(u)), *this) {
    try_([&] { super::operator=(std::forward<U>(u)); });
    return *this;
  }

  Variant& operator=(const Variant& that) {
    try_([&] { super::operator=(that); });
    return *this;
  }

  Variant& operator=(Variant&& that) noexcept(noexcept(super::operator=(std::move(that)))) {
    try_([&] { super::operator=(std::move(that)); });
    return *this;
  }

private:
  template <class F>
  void try_(F&& f) {
    try {
      std::forward<F>(f)();
    } catch (const std::bad_variant_assign& ex) {
      *this = std::nullvar;
      ex.rethrow_nested();
    }
  }
};
```

7.6 Double Storage

Using double storage to achieve strong exception safety guarantee may be a tradeoff that many people are willing to make. For those users, we can augment the proposed `variant` with another storage and implement the double storage mechanism. Again, the converse scenario would be much more difficult since there is no easy way to **remove** space from an existing object whereas augmenting one to **add** space is very easy.

```
template <class.... Types>
class Variant {
public:
  constexpr Variant() noexcept
      : which{-1}, storage{{std::nullvar, std::nullvar}} {}

  template <std::size_t I, class... Args>
  explicit constexpr Variant(
      in_place_t (&tag)(std::integral_constant<std::size_t, I>), Args&&... args)
      : which{0}, storage{{{tag, std::forward<Args>(args)...}, std::nullvar}} {}

  /* ... other in-place constructors ... */

  template <class U>  // SFINAE
  constexpr Variant(U&& u)
      : which{0}, storage{{{std::forward<U>(u)}, std::nullvar}} {}

  Variant(const Variant& that)
      : which{0}, storage{{that.storage[that.which], std::nullvar}} {}

  /* Variant(Variant&& that); */

  ~Variant() = default;

  template <size_t I, class... Args>
  void emplace(Args&&... args) {
    try_([&] {
      storage[!which].template emplace<I>(std::forward<Args>(args)...);
      storage[which] = std::nullvar;
      which = !which;
    });
  }

  /* ... other emplace ... */

  template <class U>  // SFINAE
  Variant& operator=(U&& u) {
    using T = get_best_match_t<Types..., U&&>;
    if (index() == find_index<T, Types...>{}) {
      storage[which] = std::forward<U>(u);
    } else {
      emplace<T>(std::forward<U>(u));
    }
    return *this;
  }

  /*
```

```
  Variant& operator=(const Variant& that);
  Variant& operator=(Variant&& that);
  */

  constexpr int index() const noexcept { return storage[which].index(); }

  constexpr const std::type_info& type() const noexcept {
    return storage[which].type();
  }

  void swap(Variant& that)
      noexcept(noexcept(swap(storage[which], that.storage[that.which]))) {
    swap(storage[which], that.storage[that.which]);
  }

private:
  template <class F>
  void try_(F&& f) {
    try {
      std::forward<F>(f)();
    } catch (const std::bad_variant_assign& ex) {
      ex.rethrow_nested();
    }
  }

  int which;

  std::array<std::variant<std::nullvar_t, Types...>, 2> storage;
};
```

8 Implementation

An implementation of the proposed `variant` design is available at [MPark.Variant].

9 Acknowledgements

- **Eric Friedman** and **Itay Maman** for the development of [Boost.Variant].
- **Agustín Bergé** (a.k.a K-ballo) for the development of [Eggs.Variant].
- **Axel Naumann** for writing [N4542] and its predecessors which started the discussion.
- **David Sankel** for deep design discussions and encouraging me to present my ideas during CppCon 2015.
- **Kirk Shoop** for design discussions at CppCon 2015.
- **Eric Niebler** for letting me know that submitting a proposal is a must in order to be considered for standardization.

- **Geoffrey Romer** for encouraging me to write this paper since CppCon 2014.
- **Andrei Alexandrescu** for starting the work on `variant`, writing Dr. Dobb's articles back in 2002, and for encouraging me to write this paper.
- Everyone on **std-proposals** mailing list who passionately share their thoughts, ideas, and opinions.

A special thank you to **Jason Lucas** for initially introducing me to the multi-method problem, and for the endless design and tradeoff discussions over the past few years!

10 References

[N3449] Bjarne Stroustrup, *Open and Efficient Type Switch for C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3449.pdf

[N3915] Peter Sommerlad, *apply() call a function with arguments from a tuple (V3)*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3915.pdf

[N4542] Axel Naumann, *Variant: a type-safe union (v4)*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4542.pdf

[P0050] Vicente J. Botet Escriba, *C++ generic match function*
https://github.com/viboes/tags/blob/master/doc/proposals/match/P0050.pdf

[P0051] Vicente J. Botet Escriba, *C++ generic overload functions*
https://github.com/viboes/tags/blob/master/doc/proposals/overload/P0051.pdf

[The Billion Dollar Mistake] Tony Hoare, *Null References: The Billion Dollar Mistake*
http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

[Mach7] Yuriy Solodkyy, Gabriel Dos Reis, Bjarne Stroustrup, *Mach7: Pattern Matching for C++*
https://github.com/solodon4/Mach7

[Boost.Variant] Eric Friedman, Itay Maman, *Boost.Variant*
http://www.boost.org/doc/libs/1_59_0/doc/html/variant.html

[Eggs.Variant] Agustín Bergé, *Eggs.Variant*
http://eggs-cpp.github.io/variant

[MPark.Variant] Michael Park, *Variant: Discriminated Union with Value Semantics*
https://github.com/mpark/variant

[C++ Core Guidelines] Bjarne Stroustrup, Herb Sutter *C++ Core Guidelines*
https://github.com/isocpp/CppCoreGuidelines