cppcon
the c++ conference

Enhanced Support for Value Semantics in C++17

`optional<T>   variant<Ts...>   any`

Michael Park

@mpark

@mcypark

MESOSPHERE

Outline

- Value Semantics
- `optional<T>`
- `variant<Ts...>`
- `any`
- Summary

Value Semantics

Value Semantics

	Value	Object
Characteristic	Abstract	Concrete
Identity	No	Yes
Example	`42`	`int x = 42;`

Thursday, September 28

2:00pm

Objects, Lifetimes, and References, oh my: the C++ Object Model, and Why it Matters to You

Nicole Mazzuca

What is Value Semantics?

- A model in which we operate and think in terms of **values**
- An approach to manage **objects** in a way that allows for us to adopt such a model

Strategies

- Deep-copy semantics
- Automatic lifetimes (RAII)

Some Benefits

- Closer to mathematical notation
- Referential transparency
- Avoid memory management issues

optional<T>

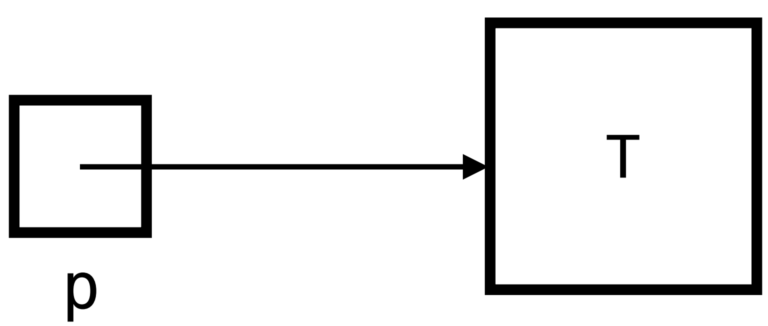
#include <optional>

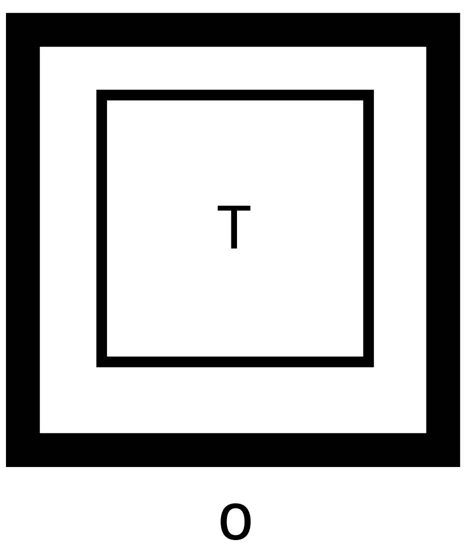
Conceptual Model

- Represents the notion of an optional object
- Models a discriminated union of `T` and `nullopt_t`
- `T *` wrapped up in a value type

```
T *p = nullptr;
p = new T(/* ... */);
```

```
optional<T> o;
o = T(/* ... */);
```





Quick Overview

```
optional<string> x = "hello";
assert(x);              // `explicit operator bool`
assert(*x == "hello");  // `operator*` (unchecked access)

optional<string> y;
assert(!y.has_value());                         // `has_value`
assert(y.value_or("world") == "world");  // `value_or`

try {
  auto s = y.value();  // `value` (checked access)
} catch (const bad_optional_access&) {}

y = x;  // assignment
assert(y != nullopt);
assert(y == x);

// `optional` invokes `string::~string` correctly.
```

Use Cases

Use Cases

- Optional Return Value
- Optional Function Parameter
- Optional Data Member

Magic Values

- A magic value is a **valid** value of type `T` used to indicate the **absence** of a value of type `T`

- Examples: `-1`, `string::npos`, `""`, `end()`

Problems

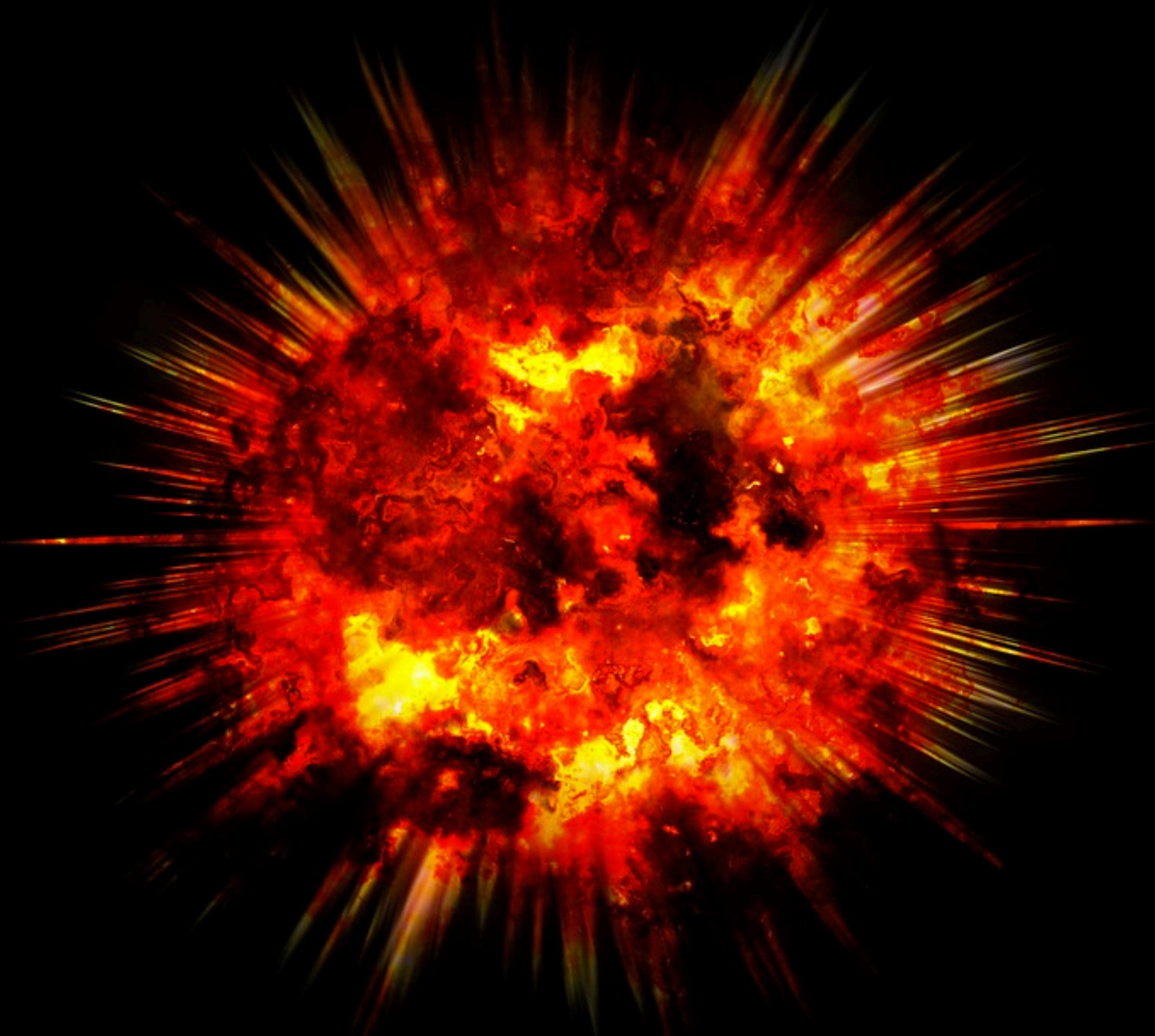
- Easy to miss, since they are not expressed in code
- There isn't always a value to be stolen (e.g., `strtol`)

Fork-Kill

```
pid_t pid = fork();
if (pid == 0) {  // child
  // ...
} else {  // parent: `pid` is child
  // ...
  kill(pid);
}
```

If *pid* equals -1, then *sig* is sent to **every** process for which the calling process has permission to send signals, except for process 1 (*init*), but see below.

http://man7.org/linux/man-pages/man2/kill.2.html



Optional Return Value

```
template <typename T>
T parse(string_view sv);
```

What if `sv` cannot be parsed into a `T`?

Solutions:

- Throw an exception
- Return a (smart) pointer to `T`
- Return a `pair<T, bool>`
- Return a `bool` and take a `T&` out-parameter

Optional Return Value

```
template <typename T>
T parse(string_view sv);
```

What if `sv` cannot be parsed into a `T`?

Solutions:

- Throw an exception
- Return a (smart) pointer to `T`
- Return a `pair<T, bool>`
- Return a `bool` and take a `T&` out-parameter

Doesn't fit well if we don't consider the inability to parse into T to be an error

Optional Return Value

```
template <typename T>
T parse(string_view sv);
```

What if `sv` cannot be parsed into a `T`?

Solutions:

- Throw an exception
- Return a (smart) pointer to `T`
- Return a `pair<T, bool>`
- Return a `bool` and take a `T&` out-parameter

We lose value semantics, and also pay for a heap allocation

Optional Return Value

```
template <typename T>
T parse(string_view sv);
```

What if `sv` cannot be parsed into a `T`?

Solutions:

- Throw an exception
- Return a (smart) pointer to `T`
- Return a `pair<T, bool>`
- Return a `bool` and take a `T&` out-parameter

`T` always needs to be constructed, and `pair<iterator, bool>` has different semantics!

Optional Return Value

```
template <typename T>
T parse(string_view sv);
```

What if `sv` cannot be parsed into a `T`?

Solutions:

- Throw an exception
- Return a (smart) pointer to `T`
- Return a `pair<T, bool>`
- Return a `bool` and take a `T&` out-parameter

`T` still always needs to be constructed, and also leads to an awkward API

Optional Return Value

```
template <typename T>
optional<T> parse(string_view sv);
```

- No exceptions
- Maintain value semantics
- No heap allocation
- `T` is only constructed if needed
- Intent is clearer
- Cleaner API

Optional Function Parameter

Before	After
`void f(Light);`	`void f(optional<Light>);`
`void g(const Heavy &) {}`	`void g(const optional<Heavy> &);` **This can be a copy!**

Optional Data Member

```
struct Person {
  string first_name;
  string last_name;
  optional<string> middle_name;
};

Person mpark = { "Chanyoung", "Park", nullopt };

// Submit some forms...

mpark.middle_name = "Michael";
```

A Few More Things...

Relational Operators

- `nullopt_t` compares less than any `T`
- All of the operators compare the engaged-ness of `optional`, then defer to the corresponding operator of `T`.
- Mixed comparisons are allowed.
  - `optional<T> == optional<U>`
  - `optional<T> == U`

Storing in Containers

```
enum class IceCreamFlavor {
  BrownSugarWithCinnamonShortBread,
  ClassicVanilla,
  CookieDoughWithPretzelsAndChocolateChips,
  EarlGreyWithMilkChocolateChips,
  SweetCornWithBerries,
  TCHOChocolate
};

map<optional<IceCreamFlavor>, int> collate(
    const vector<optional<IceCreamFlavor>> &votes) {
  for (const auto &vote : votes) {
    ++votes[vote];
  }
}
```

Optionalizing

```
class Car {
  public:
  constexpr int MAX_SPEED = 300;  // in km/h

  // Returns the current speed in km/h.

  int            get_speed() const;

  bool can_accelerate() const {
    return get_speed() < MAX_SPEED;
  }
};
```

Optionalizing

```
class Car {
  public:
  constexpr int MAX_SPEED = 300;  // in km/h

  // Returns the current speed in km/h.
  // Returns nullopt if the speedometer is non-functional.
  optional<int> get_speed() const;

  bool can_accelerate() const {
    return get_speed() < MAX_SPEED;
  }
};
```

Optionalizing

```
class Car {
  public:
  constexpr int MAX_SPEED = 300;  // in km/h

  // Returns the current speed in km/h.
  // Returns nullopt if the speedometer is non-functional.
  optional<int> get_speed() const;

  bool can_accelerate() const {
    return get_speed() < MAX_SPEED;
  }
};
```

Not a compile-time error!

`bool operator<(const optional<T> &, const U &);` is used, and `nullopt` is considered less than any `T`!

Delta from Boost.Optional

	C++17	Boost 1.65.1
Empty Tag	`nullopt`	`none`
In-Place Constructor Tag	`in_place`	`in_place_init`
Forwarding Constructor	**Yes**	**No**
Conditional Explicit	Yes	No
Reference Type Support	No	Yes
has_value();	Yes	No
operator<<	No	Yes
`T* get_ptr();`	No	Yes

`variant<Ts...>`

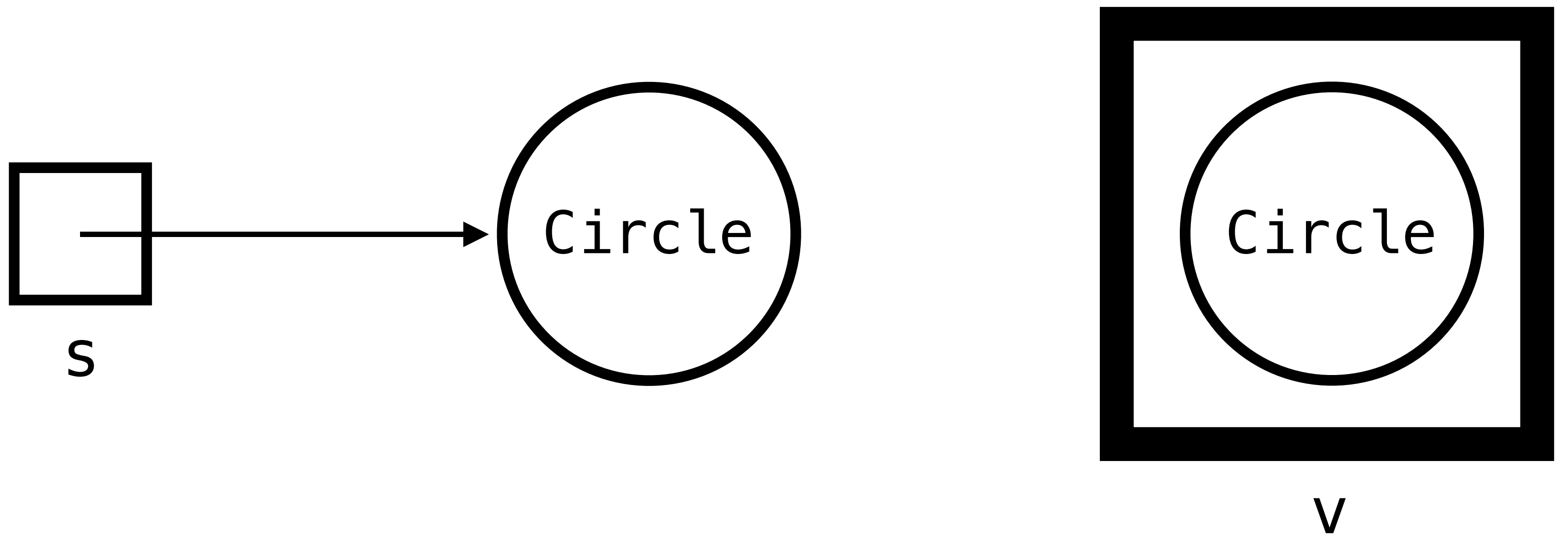
`#include <variant>`

Conceptual Model

- A type-safe `union`
- Models a discriminated union of `Ts...`
- `AbstractBase *` wrapped up in a value type

```
Shape *s =
  new Circle(/* ... */);
```

```
variant<Circle, Square> v =
  Circle(/* ... */);
```



Quick Overview

```
variant<int, string> x = "hello";
assert(holds_alternative<string>(x));  // `holds_alternative`
assert(get<string>(x) == "hello");     // `get` (checked)

variant<int, string> y;  // default-constructs to `int`
assert(y.index() == 0);  // `index`
assert(*get_if<int>(&y) == 0);  // `get_if` (checked)

try {
  auto s = get<string>(y);  // `get` (checked)
} catch (const bad_variant_access &) {}

y = x;  // assignment
assert(holds_alternative<string>(y));
assert(y == x);

// `variant` invokes `string::~string` correctly.
```

Use Cases

Use Cases

- union-like Class
- Flat / "Closed" Class Hierarchy
- Visitor Pattern

union-like Class

```
struct Cat   final { /* ... */ };
struct Dog   final { /* ... */ };
struct Horse final { /* ... */ };

struct Animal {
  enum { CAT, DOG, HORSE } kind;

  Animal(Cat   cat  ) : cat_(move(cat)), kind(CAT) {}
  Animal(Dog   dog  ) : dog_(move(dog)), kind(DOG) {}
  Animal(Horse horse) : horse_(move(horse)), kind(HORSE) {}

  ~Animal() {
    switch (kind) {
      case CAT  : cat_.~Cat();     break;
      case DOG  : dog_.~Dog();     break;
      case HORSE: horse_.~Horse(); break;
    }
  }

  union { Cat cat_; Dog dog_; Horse horse_; };
};
```

union-like Class

```
string get_sound(const Animal &animal) {
  switch (animal.kind) {
    case Animal::CAT  : /* `animal.cat_`   */ return "meow";
    case Animal::DOG  : /* `animal.dog_`   */ return "woof";
    case Animal::HORSE: /* `animal.horse_` */ return "neigh";
  }
}
```

union-like Class

```
string get_sound(const Animal &animal) {
  switch (animal.kind) {
    case Animal::CAT  : /* `animal.cat_`   */ return "meow";
    case Animal::DOG  : /* `animal.dog_`   */ return "woof";
    case Animal::HORSE: /* `animal.horse_` */ return "neigh";
  }
}
```

A LOT more code necessary for other operations such as copy/move, accessors, assignment, visitation, etc

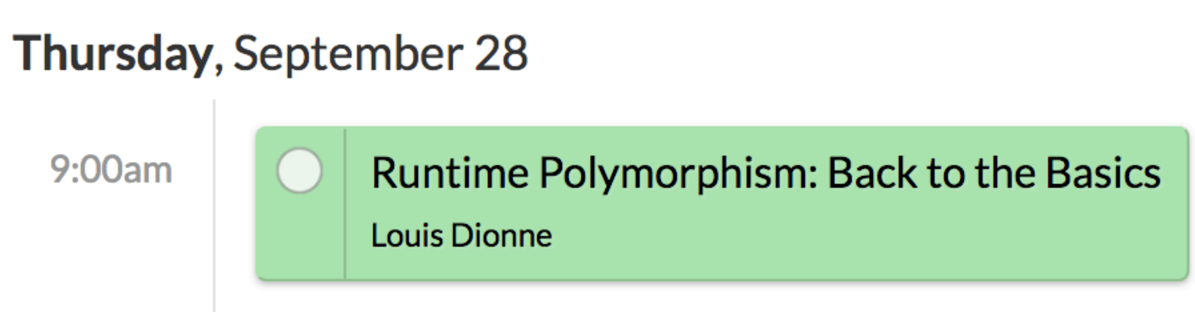
+ Value semantics
+ Non-intrusive to add new algorithms
− Error-prone due the manual pairing of `enum` and the value

Flat / "Closed" Class Hierarchy

```
struct Animal {
  virtual ~Animal() = default;
  virtual string get_sound() const = 0;
};

struct Cat   final : Animal {
  string get_sound() const override { return "meow"; }
};

struct Dog   final : Animal {
  string get_sound() const override { return "woof"; }
};

struct Horse final : Animal {
  string get_sound() const override { return "neigh"; }
};
```

Flat / "Closed" Class Hierarchy

- Lost value semantics
- Incurred dynamic allocation, and memory management
- Dual-citizenship is difficult → Multiple inheritance
- Intrusive to add new algorithms



Visitor Pattern (Ceremony)

```
struct Cat; struct Dog; struct Horse;

struct Animal {
  struct Vis {
    virtual void operator()(const Cat   &) const = 0;
    virtual void operator()(const Dog   &) const = 0;
    virtual void operator()(const Horse &) const = 0;
  };

  virtual ~Animal() = default;
  virtual void accept(const Vis &) const = 0;
};

struct Cat   final : Animal {
  void accept(const Vis &vis) const override { vis(*this); }
};

struct Dog   final : Animal {
  void accept(const Vis &vis) const override { vis(*this); }
};

struct Horse final : Animal {
  void accept(const Vis &vis) const override { vis(*this); }
};
```

Visitor Pattern (Usage)

```
string get_sound(const Animal &animal) {
  struct GetSound : Animal::Vis {
    void operator()(const Cat   &) const override {
      result = "meow";
    }
    void operator()(const Dog   &) const override {
      result = "woof";
    }
    void operator()(const Horse &) const override {
      result = "neigh";
    }

    string &result;
  };

  string result;
  animal.accept(GetSound{result});
  return result;
}
```

Visitor Pattern

+ Got back the ability to non-intrusively add algorithms
− Lots of boilerplate
− Inefficient

Variant Visitation

```
struct Cat   { /* ... */ };
struct Dog   { /* ... */ };
struct Horse { /* ... */ };

using Animal = variant<Cat, Dog, Horse>;

string get_sound(const Animal &animal) {
  struct GetSound {
    string operator()(const Cat   &) const { return "meow";  }
    string operator()(const Dog   &) const { return "woof";  }
    string operator()(const Horse &) const { return "neigh"; }
  };
  return visit(GetSound{}, animal);
}
```

What did we just do?

+ Value semantics
+ Non-intrusive to add new algorithms
+ No manual pairing of discriminator and value
+ Dual-citizenship is easy

− Code bloat

A few more things...

`valueless_by_exception()`

- An exception thrown during a type-changing operation
- ~~If all of your types are noexcept movable, you cannot get into a `valueless_by_exception` state.~~

```
struct nasty { operator int() { throw 42; } };

variant<int, float> v = 1.1f;
v.emplace<int>(nasty{});
// v.valueless_by_exception() == true
```

`valueless_by_exception()`

- There was already an exception thrown at us!
- Don't let `valueless_by_exception()` leave `catch` clauses.
- Let's not check for `valueless_by_exception()` everywhere.
  - We already don't check for `NaN` `double`s everywhere!

monostate

- Similar to `boost::blank`
- As first type, any `variant` becomes default constructible
- Unit type to add an empty state to a `variant`
- Does not change `variant`'s behavior

Forwarding Constructor

Which alternative is constructed here?

```
auto foo() { /* ... */ }

variant<T0, T1, T2> v(foo());

template <typename T>
struct id { using type = T; };

struct FUN {
  id<T0> operator()(T0) const;
  id<T1> operator()(T1) const;
  id<T2> operator()(T2) const;
};

typename invoke_result_t<FUN, decltype(foo())>::type
```

Forwarding Constructor

Which alternative is constructed here?

```
variant<string, bool> v("abc");

template <typename T>
struct id { using type = T; };

struct FUN {
  id<string> operator()(string) const;
  id<bool>   operator()(bool)   const;

};

typename invoke_result_t<FUN, decltype("abc")>::type
```

EricWF
@Eric01

Following

variant<string, bool> v = "abc" initializes the second alternative. That's boolshit.

12:23 AM - 8 Feb 2017

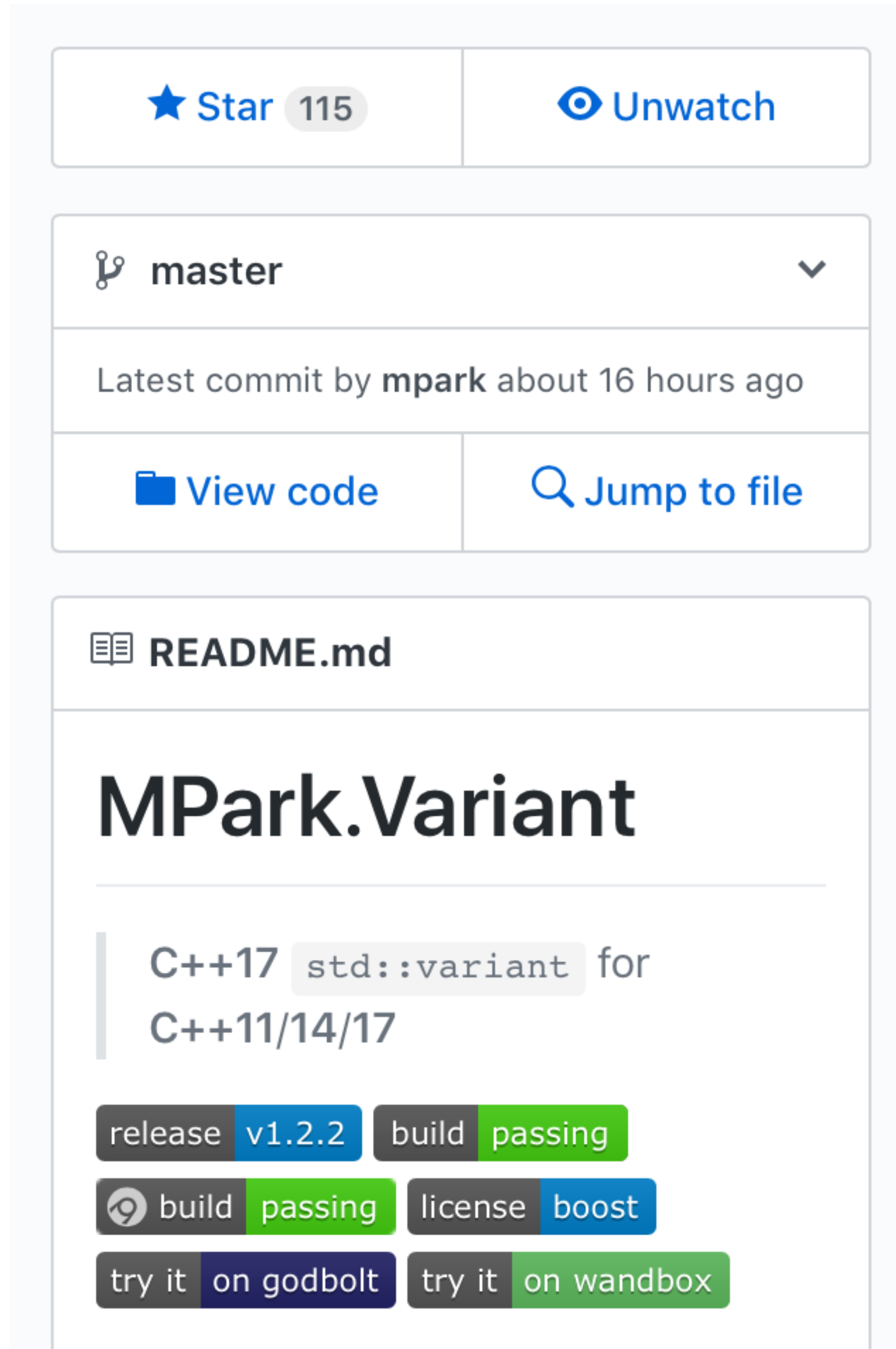
28 Retweets **66** Likes

2 28 66

Delta from Boost.Variant

	C++17	Boost 1.65.1
Empty Type	`monostate`	`blank`
Visitation	`visit`	`apply_visitor`
Non-throwing `get`	`get_if(&v)`	`get(&v)`
Dynamic-allocation during type-changing operation	**No**	**Yes**
valueless_by_exception	**Yes**	**No**
Reference Type Support	No	Yes
Index-based access	Yes	No
Special recursion support	No	Yes
In-place Constructors / `emplace`	Yes	No

Shameless Plug



`any`

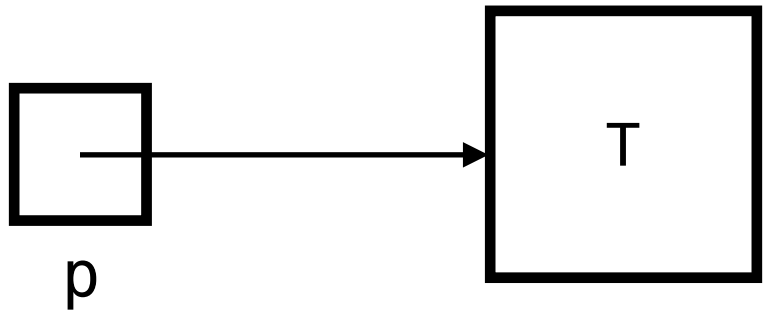
```
#include <any>
```

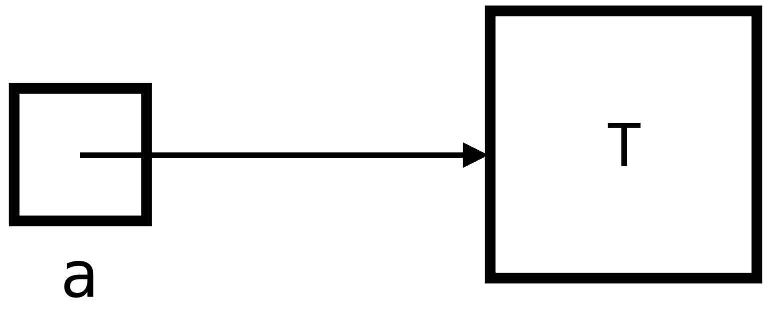
Conceptual Model

- A type-safe `void *`
- `void *` wrapped up in a value type

```
void *p = nullptr;
p = new T(/* ... */);
```

```
any a;
a = T(/* ... */);
```





What's the point then?

What's the point then?

- `any` keeps track of the type that is currently stored and check that your accesses are correct!
- Value Semantics

Quick Overview

```
any x = "hello"s;
assert(x.has_value());  // `has_value`, no `operator bool`
assert(any_cast<const string &>(x) == "hello");

any y;
assert(y.type() == typeid(void));      // `type()`
assert(any_cast<int>(&y) == nullptr);  // `any_cast(&a)`

try {
  auto i = any_cast<int>(y);  // `any_cast(a)` (checked)
} catch (const bad_any_cast &) {}

y = x;  // assignment
assert(y.type() == typeid(string));  // `type()`
// assert(y == x);  // no relational operators

// `any` invokes `string::~string` correctly.
```

Use Cases

Use Cases

- If/when a `template` won't work and `variant` cannot be used

Getting through the `virtual`

- `virtual` functions cannot be a `template`
- If the parameter or the return value needs to be *anything*

```
struct consumer {
  virtual void notify(const any &) = 0;
};
```

Delta from Boost.Any

	C++17	Boost 1.65.1
Query	`has_value()`	`empty()`
Reset	`reset`	No
In-place Constructors / `emplace`	Yes	No

Summary

Value Semantics	`optional<T>`	`variant<Ts...>`	`any`
Reference Semantics	`T *`	`AbstractBase *`	`void *`
# of Possible States	`|T| + 1`	`(... + |Ts|)`	∞

Summary

Value Semantics	`optional<T>`	`variant<Ts...>`	`any`
	Order of Preference!		
Reference Semantics	`T *`	`AbstractBase *`	`void *`
# of Possible States	`|T| + 1`	`(... + |Ts|)`	∞

Enhanced Support for Value Semantics in C++17

`optional<T>   variant<Ts...>   any`

Michael Park
@mpark
@mcypark

MESOSPHERE