Enhanced Support for Value Semantics in C++17

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

Outline

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

optional<T>

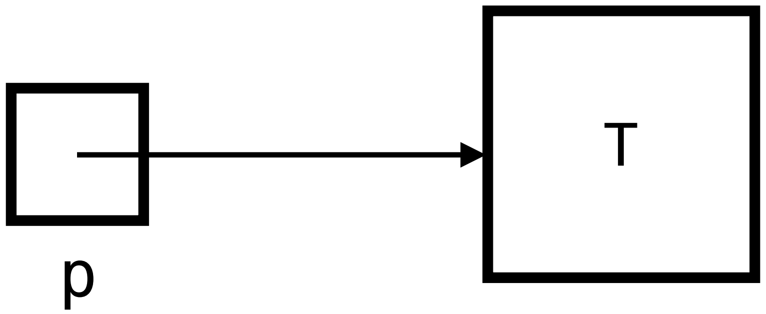
```
#include <optional>
```

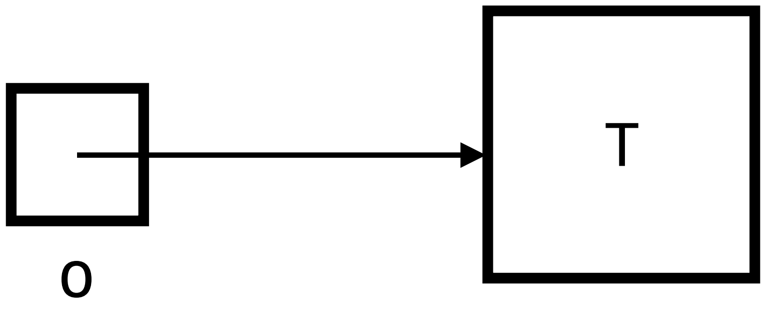
Conceptual Model

- 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<int> x = 42;
assert(x);          // `explicit operator bool`
assert(*x == 42);  // `operator*` (unchecked access)

optional<int> y;
assert(!y.has_value());          // `has_value`
assert(y.value_or(101) == 101);  // `value_or`

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

y = x;  // copy
assert(y != nullopt);
assert(y == x);
```

Use Cases

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&` 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&` 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&` 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&` 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&` 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 exception being thrown
- Maintain value semantics
- No heap allocation
- `T` does not always need to be constructed
- Intent is clearer
- Cleaner API

Avoiding Magic Values

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

  // Returns the current speed in km/h.
  // Returns -1      if the speedometer is non-functional.
  int            get_speed() const;
};
```

Avoiding Magic Values

- A magic value is a **valid** value of type `T` used to indicate the **absence** of a value of type `T`
- Some examples are `-1` or `string::npos` for an index, `""` for a `string`, and `end()` for an `iterator`
- If the values of `T` map exactly onto the range of a function, there is no value to be stolen

For example, all values of `long` are a valid result of `strtol`.
It returns `0` for variations of `"0"` **as well as** in situations where no conversion is possible.
The user must know check the status of `errno`!

Avoiding Magic Values

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

  // Returns the current speed in km/h.
  // Returns -1      if the speedometer is non-functional.
  int             get_speed() const;

  bool can_accelerate() const {

    return get_speed() < MAX_SPEED;
  }
};
```

If speedometer is non-functional, `can_accelerate() == true`!

Avoiding Magic Values

```
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 {
    optional<int> speed = get_speed();
    return speed && (*speed < MAX_SPEED);
  }
};
```

~~If speedometer is non-functional, `can_accelerate() == true`!~~

Another Example

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

Another Example

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

Pitfall: `T -> optional<T>`

```
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`!

Minor: `T -> optional<T>`

Before	After
`void f(Light);`	`void f(optional<Light>);`
`void g(const Heavy&) {}`	`void g(const optional<Heavy>&);`

Deep Dive

Requirements on `T`

- `T` shall be an object type and shall satisfy the requirements of `Destructible`.
- An *object* type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not *cv* `void`.
- A program that necessitates the instantiation of template `optional` for a reference type, or for possibly cv-qualified types `in_place_t` or `nullopt_t` is ill-formed.

Constructors

```
constexpr optional() noexcept;
constexpr optional(nullopt_t) noexcept;

template <typename U = T> EXPLICIT constexpr optional(U&&);

template <typename... Args>
constexpr explicit optional(in_place_t, Args&&...);

template <typename U, typename... Args>
constexpr explicit optional(
    in_place_t, initializer_list<U>, Args&&...);

constexpr optional(const optional&);
constexpr optional(optional&&) noexcept(see below);

template <class U> EXPLICIT optional(const optional<U>&);
template <class U> EXPLICIT optional(optional<U>&&);
```

`nullopt` Constructors

```
constexpr optional() noexcept;
constexpr optional(nullopt_t) noexcept;
```

Constructs into the `nullopt` state

```
optional<int> o;
```

`T` does not need to be `DefaultConstructible`

```
struct S { S() = delete; };
optional<S> o = nullopt;
```

Forwarding Constructor

```
template <typename U = T>
EXPLICIT constexpr optional(U&& u);
```

Direct-initializes `T` with `std::forward<U>(u)`

```
optional<int> o = 42;
```

Enabled if `is_constructible_v<T, U&&>`, and `decay_t<U>` is not `in_place_t` nor `optional<T>`

Forwarding Constructor

```
template <typename U = T>
EXPLICIT constexpr optional(U&& u);
```

Same as the one from `tuple` and `pair`.

`explicit` if and only if `!is_convertible_v<U&&, T>`

```
optional<int> x = 42;            // implicit conversion
optional<vector<int>> y = 42;    // error: no viable ctor
optional<vector<int>> z(42);     // explicit construction
```

Forwarding Constructor

```
template <typename U = T>
EXPLICIT constexpr optional(U&& u);
```

`T` is **always** *direct-initialized* (even for implicit conversions)

```
struct S {
  S(long);
  explicit S(int);
};

S s = 42;                // calls `S(long)`
optional<S> o = 42;      // calls `S(int)`
```

Only the `explicit`-ness is propagated. The behavior is not.

Forwarding Constructor

```
template <typename U = T>
EXPLICIT constexpr optional(U&& u);
```

Consider:

```
optional<vector<int>> o({1, 2});
// `U` cannot be deduced from braced-init-list!
```

Forwarding Constructor

```
template <typename U = T>
EXPLICIT constexpr optional(U&& u);
```

If a template argument has not been deduced and its corresponding template parameter has a default argument, the template argument is determined by substituting the template arguments determined for preceding template parameters into the default argument.

```
optional<vector<int>> o({1, 2});  // works!
```

In-Place Constructors

```
template <typename... Args>
constexpr explicit optional(in_place_t, Args&&... args);

template <typename U, typename... Args>
constexpr explicit optional(
    in_place_t, initializer_list<U> list, Args&&... args);
```

Direct-initializes `T` with `std::forward<Args>(args)...`

Direct-initializes `T` with `list, std::forward<Args>(args)...`

```
optional<tuple<vector<int>, string>> o(
  in_place, {1, 2, 3}, "hello");
```

Copy/Move Constructor

```
constexpr optional(const optional& that);
constexpr optional(optional&& that) noexcept(see below);
```

Copies/moves the contained value of `that`, if any.
The move constructor does NOT set `that` to `nullopt`!

```
optional<string> x = "hello";
optional<string> y(move(x));

// `x` contains a moved-out `string`!
```

Converting Constructors

```
template <class U>
EXPLICIT optional(const optional<U>& that);

template <class U>
EXPLICIT optional(optional<U>&& that);
```

Direct-initializes `T` with the contained value of `that`, if any.

`explicit` if and only if `!is_convertible_v<U (ref), T>`

```
optional<const char*> x = "hello";
optional<string> y = x;
```

Enabled if `is_constructible_v<T, U (ref)>`, and
`T` is not constructible with nor convertible from `optional<U>`
(`const`/`ref`-qualified)

Assignment

```
optional& operator=(nullopt_t) noexcept;

template <class U = T> optional& operator=(U&&);

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

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

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

template <class U> optional& operator=(const optional<U>&);
template <class U> optional& operator=(optional<U>&&);
```

Observers

```
constexpr explicit operator bool() const noexcept;
constexpr bool has_value() const noexcept;

constexpr const T& value() const&;
constexpr T& value() &;
constexpr T&& value() &&;
constexpr const T&& value() const&&;

template <class U> constexpr T value_or(U&&) const&;
template <class U> constexpr T value_or(U&&) &&;

constexpr const T* operator->() const;
constexpr T* operator->();

constexpr const T& operator*() const&;
constexpr T& operator*() &;
constexpr T&& operator*() &&;
constexpr const T&& operator*() const&&;
```

`variant<Ts...>`

`#include <variant>`

Conceptual Model

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

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

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