# std::optional<T&>

| | |
|---|---|
| Document #: | D2988R0 |
| Date: | 2023-09-30 |
| Project: | Programming Language C++ |
| Audience: | None |
| Reply-to: | Steve Downey<br><sdowney@gmail.com, sdowney2@bloomberg.net> |


# 1 Abstract

An optional over a reference such that the post condition on assignment is independent of the engaged state, always producing a rebound reference, and assigning a U to a T is disallowed by static_assert if a bind a U can not be bound to a T&.


# 2 Comparison table

**Before**

```
std::shared_ptr<Cat> cat = find_cat("Fido");
// or
std::map<std::string, Cat>::iterator cat
    = find_cat("Fido");
// or
Cat* cat = find_cat("Fido");
```

**After**

```
std::optional<Cat&> cat = find_cat("Fido");
```

**Before**

```
std::optional<Cat*> c = find_cat("Fido");
if (c) {
    if (*c) {
        *c = Cat("Fynn", color::orange);
    }
}
```

**After**

```
std::optional<Cat&> c = find_cat("Fido");
if (c) {
    *c = Cat("Fynn", color::orange);
}
//or
o.transform([&](auto c&){
    c = Cat("Fynn", color::orange);
    });
```

# 3 Motivation

Optionals holding references are common other than in the standard libary's implementation. The desire for such a feature is well understood, and many optional types in commonly used libraries provide it, with the semanics proposed here.

The research in JeanHeyd Meneide's *References for Standard Library Vocabulary Types - an optional case study.* [P1683R0] shows conclusively that rebind semantics are the only safe semantic as assign through on engaged is too bug-prone. Implementations that attempt assign-through are abandoned. The standard library should follow existing practice and supply an `optional<T&>` that rebinds on assignment.

There is a principled reason not to provide a partial specialization over `T&` as the sematics are in some ways subtly different than the primary template. Assignment may have side-effects not present in the primary, which has pure value semantics. However, I argue this is misleading, as reference semantics often has side-effects. The proposed semantic is similar to what an `optional<std::reference_wrapper<T>>` provides, with much greater usability.

There are well motivated suggestions that perhaps instead of an `optional<T&>` there should be an `optional_ref<T>` that is an independent primary template. This proposal rejects that. We need a policy over all sum types as to how reference semantics should work, as optional is a variant over T and monostate. That the library sum type can not express the same range of types as the product type, tuple, is an increasing problem as we add more types logically equivalent to a variant. The template types `optional` and `expected` should behave as extensions of `variant<T, monostate>` and `variant<T, E>`, or we lose the ability to reason about generic types.

That from `std::tuple<Args...>` we can't guarantee that `std::variant<Args...>` is valid is a problem, and one that reflection can't solve. A language sum type could, but we need agreement on the semantics.

The sematics of a variant with a reference are as if it holds the address of the referent when refering to that referent. All other sematics are worse. Not being albe to express a variant<T&> is inconsistent, hostile, and strictly worse than disallowing it.

# 4 Design

The design is straightforward. The optional<T&> holds a pointer to the underlying object of type T, or nullptr if the optional is disengaged. The implementation is simple, especially with C++20 and up techniques, using concept constraints. As the held pointer is a primitive regular type with reference semantics, many operations can be defaulted and are noexcept by nature. See https://github.com/steve-downey/optional_ref and https://github.com/steve-downey/optional_ref/blob/main/src/smd/optional/optional.h for a reference implementation. The optional<T&> implementation is less than 200 lines of code, much of it the monadic functions with identical textual implementations with different signatures and different overloads being called.

# 5 Shallow vs Deep `const`

There is some implementation divergence in optionals about deep const for `optional<T&>`. That is, can the referred to `int` be modified through a `const optional<int&>`. Does `operator->()` return an `int*` or a `const int*`, and does `operator*()` return an `int&` or a `const int&`. I believe it is overall more defensible if the `const` is shallow as it would be for a `struct ref {int * p;}` where the constness of the struct ref does not affect if the p pointer can be written through. This is consistent with the rebinding behavior being proposed.

Where deeper constness is desired, `optional<const T&>` would prevent non const access to the underlying object.

# 6 Wording

Modify 22.5 Optional Objects

add

```
Class template optional[optional.optional_ref]
General[optional.optional_ref.general]

namespace std {
namespace std {
  template<class T>
```

```
class optional<T&> {
public:
  using value_type = T;
  [optional_ref.ctor], constructors
      constexpr optional() noexcept;
      constexpr optional(nullopt_t) noexcept;
      constexpr optional(const optional&) noexcept;
      constexpr optional(optional&&) noexcept;
      template<class U = T>
        constexpr optional(U&&);
      template <class U>
        constexpr explicit optional(const optional<U>& rhs) noexcept;

    [optional_ref.dtor], destructor
      constexpr ~optional();

    [optional_ref.assign], assignment
      constexpr optional& operator=(nullopt_t) noexcept;
      constexpr optional& operator=(const optional&);
      constexpr optional& operator=(optional&&) noexcept(/* see below */);
      template <class U = T>
        constexpr optional& operator=(U&&);
      template <class U>
        constexpr optional& operator=(const optional<U>&);
      template <class U>
        constexpr optional& operator=(optional<U>&&);

    [optional_ref.swap], swap
      constexpr void swap(optional&) noexcept(/* see below */);

    [optional_ref.observe], observers
      constexpr T*        operator->() const noexcept;
      constexpr T&        operator*() const& noexcept;
      constexpr T&&       operator*() const&& noexcept;
      constexpr explicit  operator bool() const noexcept;
      constexpr bool      has_value() const noexcept;
      constexpr T&        value() const&;
      constexpr T&& value() const&&;
      template <class U>
        constexpr T value_or(U&&) const&;

    [optional_ref.monadic], monadic operations
      template <class F>
        constexpr auto and_then(F&& f) &;
      template <class F>
        constexpr auto and_then(F&& f) &&;
      template <class F>
        constexpr auto and_then(F&& f) const&;
      template <class F>
        constexpr auto and_then(F&& f) const&&;
      template <class F>
        constexpr auto transform(F&& f) &;
      template <class F>
```

```
        constexpr auto transform(F&& f) &&;
      template <class F>
        constexpr auto transform(F&& f) const&;
      template <class F>
        constexpr auto transform(F&& f) const&&;
      template <class F>
        constexpr optional or_else(F&& f) &&;
      template <class F>
        constexpr optional or_else(F&& f) const&;

    [optional_ref.mod], modifiers
      constexpr void reset() noexcept;
  private:
    T *val;         // exposition only
```

Constructors[optional_ref.ctor]

```
constexpr optional() noexcept;

constexpr optional(nullopt\_t) noexcept;
```

1 *Postconditions*: *this does not contain a value.

2 *Remarks*: No contained value is initialized. For every object type T these constructors are constexpr constructors ([dcl.constexpr]).

```
constexpr optional(const optional& rhs);
```

3 *Effects*: Initializes val with the value of rhs.val

4 *Postconditions*: rhs.has_value() == this->has_value().

5 *Remarks*: The constructor is trivial.

```
constexpr optional(optional&&) noexcept;
```

3 *Effects*: Initializes val with the value of rhs.val

4 *Postconditions*: rhs.has_value() == this->has_value().

5 *Remarks*: The constructor is trivial.

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

3 *Constraints*:

(3.1) – !is-optional<decay_t<U>>::value is true

3 *Mandates*: (3.1) – std::is_constructible_v<std::add_lvalue_reference_t<T>, (3.1) U>; – std::is_lvalue_reference<U>::value

3 *Effects*: Initializes val with the address of u

4 *Postconditions*: this->has_value() == true.

```
      template <class U>
        constexpr explicit optional(const optional<U>& rhs) noexcept;
```

3 *Constraints*:

(3.1) – !is-optional<decay_t<U>>::value is true

3 *Mandates*: (3.1) – std::is_constructible_v<std::add_lvalue_reference_t<T>, (3.1) U>; – std::is_lvalue_reference<U>::value

3 *Effects*:

Destructor [optional_ref.dtor]

```
        constexpr ~optional();
```

5 *Remarks*: The destructor is trivial.

# 7 References

[P1683R0] JeanHeyd Meneide. 2020-02-29. References for Standard Library Vocabulary Types - an optional case study.
https://wg21.link/p1683r0