We propose to fix a hole intentionally left in std::optional —
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 U can not be bound to a T&.

Contents

1 Comparison table 1
2 Motivation 2
3 Design 3
4 Shallow vs Deep const 3
5 Proposal 3
6 Wording 3
7 Impact on the standard 9

Changes Since Last Version

— Changes since R0,
  — Wording update

1 Comparison table

1.1 Using a raw pointer result for an element search function
This is the convention the C++ core guidelines suggest, to use a raw pointer for representing optional non-owning references. However, there is a user-required check against ‘nullptr’, no type safety meaning no safety against mis-interpreting such a raw pointer, for example by using pointer arithmetic on it.

```cpp
Cat* cat = find_cat("Fido");
if (cat!=nullptr) { return doit(*cat); }
std::optional<Cat&> cat = find_cat("Fido");
return cat.and_then(doit);
```
1.2 returning result of an element search function via a (smart) pointer

The disadvantage here is that std::experimental::observer_ptr<T> is both non-standard and not well named, therefore this example uses shared_ptr that would have the advantage of avoiding dangling through potential lifetime extension. However, on the downside is still the explicit checks against the nullptr on the client side, failing so risks undefined behavior.

```cpp
std::shared_ptr<Cat> cat = find_cat("Fido");
if (cat != nullptr) {
    /* ... */
}
```

1.3 returning result of an element search function via an iterator

This might be the obvious choice, for example, for associative containers, especially since their iterator stability guarantees. However, returning such an iterator will leak the underlying container type as well necessarily requires one to know the sentinel of the container to check for the not-found case.

```cpp
std::map<std::string, Cat>::iterator cat = find_cat("Fido");
if (cat != theunderlyingmap.end()) {
    /* ... */
}
```

1.4 Using an optional<T*> as a substitute for optional<T&>

This approach adds another level of indirection and requires two checks to take a definite action.

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

2 Motivation

Other than the standard library’s implementation of optional, optionals holding references are common. The desire for such a feature is well understood, and many optional types in commonly used libraries provide it, with the semantics proposed here. One standard library implementation already provides an implementation of std::optional<T&> but disables its use, because the standard forbids it.

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.

Additional background reading on optional<T&> can be found in JeanHeyd Meneide’s article _To Bind and Loose a Reference_ [REFBIND].

In freestanding environments or for safety-critical libraries, an optional type over references is important to implement containers, that otherwise as the standard library either would cause undefined behavior when accessing an non-available element, throw an exception, or silently create the element. Returning a plain pointer for such an optional reference, as the core guidelines suggest, is a non-type-safe solution and doesn’t protect in any way from accessing an non-existing element by a nullptr de-reference. In addition, the monadic APIs of std::optional makes is especially attractive by streamlining client code receiving such an optional reference, in contrast to a pointer that requires an explicit nullptr check and de-reference.

There is a principled reason not to provide a partial specialization over T& as the semantics 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, because 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 we can’t guarantee from std::tuple<Args...> (product type) that std::variant<Args...> (sum type) 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 semantics of a variant with a reference are as if it holds the address of the referent when referring to that referent. All other semantics are worse. Not being able to express a variant<T&> is inconsistent, hostile, and strictly worse than disallowing it.

Thus, we expect future papers to propose std::expected<T&, E> and std::variant with the ability to hold references. The latter can be used as an iteration type over std::tuple elements.

3 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 [Downey_smd_optional_optional_T] and [rawgitus8:online]. 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.

In place construction is not supported as it would just be a way of providing immediate life-time issues.

4 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.

5 Proposal

Add a reference specialization for the std::optional template.

6 Wording

нул.1 Class template optional
нул.1.1 General

namespace std {
    template <class T>
    class optional<T&> {
    public:
        using value_type = T&;

        //нул.1.2 constructors
        constexpr optional() noexcept;
        constexpr optional(nullopt_t) noexcept;
        constexpr optional(const optional&);
    }
```cpp
constexpr optional(optional&&) noexcept(see below);

template<class U = T>
constexpr explicit(see below) optional(U&&);

template<class U>
constexpr explicit(see below) optional(const optional<U>&);

template<class U>
constexpr explicit(see below) optional(optional<U>&&);

// 1.3 destructor
constexpr ~optional();

// 1.4 assignment
constexpr optional& operator=(nullopt_t) noexcept;
constexpr optional& operator=(const optional&);
constexpr optional& operator=(optional&);  // see below

// 1.4 swap
constexpr void swap(optional&) noexcept;

// 1.5 observers
constexpr const T* operator->() const noexcept;
constexpr T* operator->() noexcept;
constexpr const T& operator*() const & noexcept;
constexpr T& operator*() & noexcept;
constexpr T&& operator*() && noexcept;
constexpr const T&& operator*() const && noexcept;
constexpr bool has_value() const noexcept;
constexpr explicit operator bool() const noexcept;

// 1.6, monadic operations
```

```cpp
template<class F> constexpr auto and_then(F&& f) &;
template<class F> constexpr auto and_then(F&& f) &&;
```

```cpp
template<class F> constexpr auto transform(F&& f) &;
```

```cpp
template<class F> constexpr optional or_else(F&& f) &&;
```

```cpp
private:
T *val; // exposition only
};
```

1 Any instance of optional<T&> at any given time either refers to a value or does not refer to a value. When an instance of optional<T&> refers to a value, it means that an object of type T, referred to as the optional object’s
referred to value, is pointed to from the storage of the optional object. When an object of type `optional<T&>` is contextually converted to `bool`, the conversion returns `true` if the object refers to a value; otherwise the conversion returns `false`.

2 When an `optional<T&>` object refers to a value, member `val` points to the referred to object.

### 1.2 Constructors

The exposition-only variable template `converts-from-any-cvref` is used by some constructors for `optional`.

```cpp
template<class T, class W>
constexpr bool converts-from-any-cvref = // exposition only
disjunction_v<is_constructible<T, W&>, is_convertible<W&, T>,
is_constructible<T, const W&>, is_convertible<const W&, T>,
is_constructible<T, W>, is_convertible<W, T>>;
```

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

Postconditions: *this does not refer to a value.

Remarks: For every object type `T` these constructors are `constexpr` constructors (??).

```cpp
constexpr optional(const optional& rhs) noexcept = default;
```

Effects: If `rhs` refers to a value, direct-non-list-initializes the referred to value with `rhs.val`.

Postconditions: `rhs.has_value() == this->has_value()`.

Throws: Nothing

```cpp
constexpr optional(optional&& rhs) noexcept = default;
```

Effects: If `rhs` refers to a value, direct-non-list-initializes the referred to value with `rhs.val`. `rhs.has_value()` is unchanged.

Postconditions: `rhs.has_value() == this->has_value()`.

```cpp
template<class U = T> constexpr explicit(see below) optional(U&& v);
```

Constraints:

- `(9.1) is_same_v<remove_cvref_t<U>, optional> is false, and`
- `(9.2) if T is cv bool, remove_cvref_t<U> is not a specialization of optional.`

Mandates:

- `(10.1) is_constructible_v<std::add_lvalue_reference_t<T>, U> is true`
- `(10.2) is_lvalue_reference<U>::value> is true`

Effects: Direct-non-list-initializes the referred to value with `std::addressof(v)`.

Postconditions: *this refers to a value.

```cpp
template<class U> constexpr explicit(see below) optional(const optional<U>& rhs);
```

Constraints:

- `(13.1) is_same_v<remove_cvref_t<U>, optional> is false, and`
- `(13.2) if T is cv bool, remove_cvref_t<U> is not a specialization of optional.`

Mandates:

- `(14.1) is_constructible_v<std::add_lvalue_reference_t<T>, U> is true`
- `(14.2) is_lvalue_reference<U>::value> is true`
Effects: If \( \text{rhs} \) refers to a value, direct-non-list-initializes the referred to value with `addressof(\text{rhs}.\text{value()})`.

Postconditions: \( \text{rhs}.\text{has\_value()} == \text{this}->\text{has\_value()} \).

### 1.3 Destructor

```cpp
constexpr ~optional() = default;
```

### 1.4 Assignment

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

Postconditions: \( \ast\text{this} \) does not contain a value.

Returns: \( \ast\text{this} \).

```cpp
optional<T>& operator=(const optional& rhs) noexcept = default;
```

Postconditions: \( \text{rhs}.\text{has\_value()} == \text{this}->\text{has\_value()} \).

Returns: \( \ast\text{this} \).

```cpp
optional& operator=(optional&& rhs) noexcept = default;
```

Postconditions: \( \text{rhs}.\text{has\_value()} == \text{this}->\text{has\_value()} \).

Returns: \( \ast\text{this} \).

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

Constraints:

- (7.1) \( \text{is\_same_v<remove\_cvref\_t<\text{T}>, optional>} \) is false, and
- (7.2) if \( T \) is `cv bool`, `remove\_cvref\_t<\text{T}>` is not a specialization of `optional`.

Mandates:

- (8.1) `is\_constructible_v<std::add\_lvalue\_reference\_t<T>`, \( U \)` is true
- (8.2) `is\_lvalue\_reference<\text{U}>::\text{value}` is true

Effects: Assigns the referred to value with `std::addressof(v)`.

Postconditions: \( \ast\text{this} \) refers to a value.

```cpp
template<class U = T> constexpr optional<T>& operator=(const optional<U>& rhs);
```

Mandates:

- (11.1) `is\_constructible_v<std::add\_lvalue\_reference\_t<T>`, \( U \)` is true
- (11.2) `is\_lvalue\_reference<\text{U}>::\text{value}` is true

Postconditions: \( \text{rhs}.\text{has\_value()} == \text{this}->\text{has\_value()} \).

Returns: \( \ast\text{this} \).

```cpp
template <class U = T> optional& emplace(U&& u) noexcept;
```

Constraints:

- (13.1) `is\_same_v<remove\_cvref\_t<\text{U}>, optional>` is false, and
- (13.2) if \( T \) is `cv bool`, `remove\_cvref\_t<\text{U}>` is not a specialization of `optional`.

Effects: Assigns \( \ast\text{this} `std::forward<\text{U}>(\text{u}) \)

Postconditions: \( \ast\text{this} \) refers to a value.
1.5 Swap

```cpp
constexpr void swap(optional& rhs) noexcept;
```

**Effects:** *this and *rhs will refer to each others initial referred to objects.

1.6 Observers

```cpp
constexpr T* operator->() const noexcept;
```

**Returns:** val.

**Remarks:** These functions areconstexpr functions.

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

**Preconditions:** *this refers to a value.

**Returns:** *val.

**Remarks:** These functions areconstexpr functions.

```cpp
constexpr bool has_value() const noexcept;
```

**Returns:** true if and only if *this refers to a value.

**Remarks:** This function is aconstexpr function.

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

**Effects:** Equivalent to:

```cpp
if (has_value())
    return *value_;
    throw bad_optional_access();
```

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

**Effects:** Equivalent to:

```cpp
if (has_value())
    return *value_;
    throw bad_optional_access();
```

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

**Mandates:** is_copy_constructible_v<T> && is_convertible_v<U&&, T> is true.

**Effects:** Equivalent to:

```cpp
return has_value() ? value() : std::forward<U>(u));
```

```cpp
template<class U> constexpr T value_or(U&& v) &&;
```
Mandates: `is_move_constructible_v<T> && is_convertible_v<U&&, T>` is true.

Effects: Equivalent to:

```cpp
define return has_value() ? value() : static_cast<T>(std::forward<U>(u));
```

### 1.7 Monadic operations

> [optional.monadic]

```cpp
template<class F> constexpr auto and_then(F&& f) &;
```

Let `U` be `invoke_result_t<F, T&>`.

Mandates: `remove_cvref_t<U>` is a specialization of `optional`.

Effects: Equivalent to:

```cpp
define return has_value() ? std::invoke(std::forward<F>(f), value()) : result(nullopt);
```

```cpp
template<class F> constexpr auto transform(F&& f) &;
```

Let `U` be `remove_cv_t<invoke_result_t<F, T&>>`.

Mandates: `U` is a non-array object type other than `in_place_t` or `nullopt_t`. The declaration

```cpp
U u(invoke(std::forward<F>(f), *val));
```

is well-formed for some invented variable `u`.

[Note 1: There is no requirement that `U` is movable (??). — end note]

Returns: If `*this` refers to a value, an `optional<U>` object whose referred to value is the result of

```cpp
define invoke(std::forward<F>(f), *val));
```

```cpp
template<class F> constexpr auto transform(F&& f) &&;
```

Let `U` be `remove_cv_t<invoke_result_t<F, T&>>`.

Mandates: `U` is a non-array object type other than `in_place_t` or `nullopt_t`. The declaration

```cpp
U u(invoke(std::forward<F>(f), std::move(*val)));
```

is well-formed for some invented variable `u`.

[Note 2: There is no requirement that `U` is movable (??). — end note]

Returns: If `*this` refers to a value, an `optional<U>` object whose referred to value is the result of

```cpp
define invoke(std::forward<F>(f), std::move(*val));
```

Constraints: `F` models `invocable<>` and
Mandates: is_same_v<remove_cvref_t<invoke_result_t<F>>, optional> is true.

Effects: Equivalent to:
```cpp
if (*this) {
    return *this;
} else {
    return std::move(*this);
}
```

template<class F> constexpr optional or_else(F&& f) &&;

Constraints: F models invocable<> and
Mandates: is_same_v<remove_cvref_t<invoke_result_t<F>>, optional> is true.

Effects: Equivalent to:
```cpp
if (*this) {
    return std::move(*this);
} else {
    return std::forward<F>(f)();
}
```

1.8 Modifiers [optional.mod]

constexpr void reset() noexcept;

Postconditions: *this does not refer to a value.

7 Impact on the standard

A pure library extension, affecting no other parts of the library or language.
The proposed changes are relative to the current working draft [N4910].

Document history

— Changes since R0
   — Wording Updates

References


