std::optional<T&>

Steve Downey <sdowney@gmail.com>
Peter Sommerlad <peter.cpp@sommerlad.ch>

Document #: P2988R4
Date: 2024-04-16
Project: Programming Language C++
Audience: LEWG

Abstract

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

Changes Since Last Version

— **Changes since R3**
  — make_optional discussion - always value
  — value_or discussion - always value

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.

```
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.

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

```
std::optional<Cat&> cat = find_cat("Fido");
cat.and_then([](Cat& thecat){/* ... */}
```

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.

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

```
std::optional<Cat&> cat
        = find_cat("Fido");
cat.and_then([](Cat& thecat){/* ... */}
```

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.

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

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

//or

o.transform([](Cat& c){
  c = 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 [rawgithu58: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.

3.1 Relational Operations

The definitions of the relational operators are the same as for the base template. Interoperable comparisons between T and optional<T&> work as expected. This is not true for the boost optional<T&>.

3.2 make_optional

Because of existing code, `make_optional<T&>` must return optional<T> rather than optional<T&>. Returning optional<T&> is consistent and defensible, and a few optional implementations in production make this choice. It is, however, quite easy to construct a make_optional expression that deduces a different category causing possibly dangerous changes to code.

There was some discussion about using library technology to allow selection of the reference overload via the literal spelling `make_optional<int&>`. There was anti-consensus to do so. There are exising instances of that spelling that today return an `optional<T>`, although it is very likely these are mistakes or possibly other optionals. The spelling `optional<T&>{}` is acceptable as there is no multi-argument emplacement version as there is no location to construct such an instance.

There was also discussion of using `std::referenece_wrapper` to indicate reference use, in analogy with std::tuple. Unfortunately there are existing uses of optional over reference_wrapper as a workaround for lack of reference specialiazation, and it would be a breaking change for such code.

3.3 Trivial construction

Construction of `optional<T&>` should be trivial, because it is straightforward to implement, and `optional<T>` is trivial. Boost is not.

3.4 Value Category Affects value()

For several implementations there are distinct overloads for functions depending on value category, with the same implementation. However, this makes it very easy to accidentally steal from the underlying referred to object. Value category should be shallow. Thanks to many people for pointing this out. If "Deducing `this`" had been used, the problem would have been much more subtle in code review.

3.5 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.

3.6 Conditional Explicit

As in the base template, `explicit` is made conditional on the type used to construct the optional. `explicit(!std::is_convertible_v<U, T>)`. This is not present in boost::optional, leading to differences in construction between braced initialization and = that can be surprising.

3.7 value_or

~~Have `value_or` return a `T&`. Check that the supplied value can be bound to a `T&`.~~

After extensive discussion, it seems there is no particularly wonderful solution for `value_or` that does not involve a time machine. Implemntations of optionals that support reference semantics diverge over the return type, and the current one is arguabley wrong, and should use something based on `common_reference_type`, which of course did not exist when `optional` was standardized.

The weak consensus is to return a `T` from `optional<T&>::value_or` as this is least likely to cause issues. There was at least one strong objection to this choice, but all other choices had more objections. The author indends to propose free functions `reference_or`, `value_or`, `or_invoke`, and `yield_if` over all types modeling optional-like, `concept std::maybe`, in the next revision of [**?**]. This would cover `optional`, `expected`, and pointer types.

Having `value_or` return by value also allows the common case of using a literal as the alternative to be expressed concisely.

3.8 Compiler Explorer Playground

See `https://godbolt.org/z/n5oooK58W` for a playground with relevant Google Test functions and various optional implementations made available for cross reference.

4 Proposal

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

5 Wording

𝔉.𝔉.1 Class template `optional` [optional.optional]

𝔉.𝔉.1.1 General [optional.optional.general]

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

    // 𝔉.𝔉.1.2, constructors
    constexpr optional() noexcept;
    constexpr optional(nullopt_t) noexcept;
    constexpr optional(const optional&) = default;
    constexpr optional(optional&&) noexcept = default;
    template<class U>
    constexpr explicit(see below) optional(U&&);
    template<class U>
    constexpr explicit(see below) optional(const optional<U>&);
```

```
    // ?.?.1.3, destructor
    constexpr ~optional();

    // ?.?.1.4, assignment
    constexpr optional& operator=(nullopt_t) noexcept;
    constexpr optional& operator=(const optional&) noexcept = default;
    constexpr optional& operator=(optional&&) noexcept = default;
    template<class U = T> constexpr optional& operator=(U&&);
    template<class U> constexpr optional& operator=(const optional<U>&);

    template<class U> constexpr T& emplace(U&&);

    // ?.?.1.5, swap
    constexpr void swap(optional&) noexcept;

    // ?.?.1.6, observers
    constexpr T* operator->() const noexcept;
    constexpr T& operator*() const noexcept;
    constexpr explicit operator bool() const noexcept;
    constexpr bool has_value() const noexcept;
    constexpr const T& value() const;                          // freestanding-deleted
    template<class U> constexpr T value_or(U&&) const;

    // ?.?.1.7, monadic operations
    template<class F> constexpr auto and_then(F&& f) const;
    template<class F> constexpr auto transform(F&& f) const;
    template<class F> constexpr optional or_else(F&& f) const;

    // ?.?.1.8, modifiers
    constexpr void reset() noexcept;

  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 [optional.ctor]

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

1 *Postconditions*: `*this` does not refer to a value.

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

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

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

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

5 *Throws*: Nothing

6 *Remarks*: This constructor is trivial.

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

7 *Effects*: Initializes `val` with the value of `rhs.val`. `rhs.has_value()` is unchanged.

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

9 *Remarks*: This constructor is trivial.

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

10 *Constraints*:

(10.1) — `is_same_v<remove_cvref_t<U>, optional>` is `false`, and

(10.2) — if `T` is *cv* `bool`, `remove_cvref_t<U>` is not a specialization of `optional`.

11 *Mandates*:

(11.1) — `is_constructible_v<add_lvalue_reference_t<T>, U>` is `true`

(11.2) — `is_lvalue_reference<U>::value>` is `true`

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

13 *Postconditions*: `*this` refers to a value.

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

14 *Constraints*:

(14.1) — `is_same_v<remove_cvref_t<U>, optional>` is `false`.

15 *Effects*: If `rhs` refers to a value, initializes `val` with the value of`addressof(rhs.value()))`.

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

17 *Remarks*: The expression inside `explicit` is equivalent to:

```
!is_convertible_v<U, T>
```

𝟐.𝟐.1.3 Destructor [optional.dtor]

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

1 *Remarks*: This destructor is trivial.

𝟐.𝟐.1.4 Assignment [optional.assign]

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

1 *Postconditions*: `*this` does not refer to a value.

2 *Returns*: `*this`.

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

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

4 *Returns*: `*this`.

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

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

6 *Returns*: `*this`.

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

7 *Constraints*:

(7.1) — `is_same_v<remove_cvref_t<U>, optional>` is `false`

(7.2) — `conjunction_v<is_scalar<T>, is_same<T, decay_t<U>>>` is `false`

8 *Mandates*:

(8.1) — `is_constructible_v<add_lvalue_reference_t<T>, U>` is `true`

(8.2) — `is_lvalue_reference<U>::value>` is `true`

9 *Effects*: Assigns the `val` the value of `addressof(v)`.

10 *Postconditions*: `*this` refers to a value.

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

11 *Mandates*:

(11.1) — `is_constructible_v<add_lvalue_reference_t<T>, U>` is `true`

(11.2) — `is_lvalue_reference<U>::value>` is `true`

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

13 *Returns*: `*this`.

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

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`.

14 *Effects*: Assigns `*this forward<U>(u)`

15 *Postconditions*: `*this` refers to a value.

𝔉.𝔉.1.5 Swap [optional.swap]

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

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

𝔉.𝔉.1.6 Observers [optional.observe]

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

1 *Returns*: `val`.

2 *Remarks*: These functions are constexpr functions.

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

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

4 *Returns*: `*val`.

5 *Remarks*: These functions are constexpr functions.

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

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

7 *Remarks*: This function is a constexpr function.

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

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

9 *Remarks*: This function is a constexpr function.

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

10 *Effects*: Equivalent to:

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

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

11 *Mandates*: `is_constructible_v<add_lvalue_reference_t<T>, decltype(v)>` is `true`.

12 *Effects*: Equivalent to:

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

𝔵.𝔵.1.7 Monadic operations [optional.monadic]

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

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

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

3 *Effects*: Equivalent to:

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

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

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

5 *Returns*: If `*this` refers to a value, an `optional<U>` object whose refered to value is the result of `invoke(forward<F>(f), *val)`; otherwise, `optional<U>()`.

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

6 *Mandates*: `is_same_v<remove_cvref_t<invoke_result_t<F>>, optional>` is `true`.

7 *Effects*: Equivalent to:

```
if (has_value()))
  return value()
else
  return forward<F>(f)();
}
```

𝔵.𝔵.1.8 Modifiers [optional.mod]

```
constexpr void reset() noexcept;
```

1 *Postconditions*: `*this` does not refer to a value.

6 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 R1**

  — Design points called out

— **Changes since R0**

  — Wording Updates

References

[Downey_smd_optional_optional_T] Stephen Downey. optional<T&>. https://github.com/steve-downey/optional_ref.

[N4910] Thomas Köppe. N4910: Working draft, standard for programming language c++. https://wg21.link/n4910, 3 2022.

[P1683R0] JeanHeyd Meneide. P1683R0: References for standard library vocabulary types - an optional case study. https://wg21.link/p1683r0, 2 2020.

[REFBIND] JeanHeyd Meneide. To bind and loose a reference | the pasture. https://thephd.dev/to-bind-and-loose-a-reference-optional. (Accessed on 01/01/2024).

[rawgithu58:online] Stephen Downey. raw.githubusercontent.com/steve-downey/optional_ref/main/src/smd/optional/optional.h. https://raw.githubusercontent.com/steve-downey/optional_ref/main/src/smd/optional/optional.h. (Accessed on 01/01/2024).

Changes Log

— **Changes since R1**

  — Design points called out

Implementation

```cpp
namespace detail {

template <class T>
struct is_optional_impl : std::false_type {};

template <class T>
struct is_optional_impl<optional<T>> : std::true_type {};

template <class T>
using is_optional = is_optional_impl<std::decay_t<T>>;

} // namespace detail

/****************/
/* optional<T&> */
/****************/

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

  private:
    T* value_; // exposition only

  public:
    //  \rSec3[optional.ctor]{Constructors}

    constexpr optional() noexcept : value_(nullptr) {}

    constexpr optional(nullopt_t) noexcept : value_(nullptr) {}

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

    template <class U = T>
        requires(!detail::is_optional<std::decay_t<U>>::value)
    constexpr explicit(!std::is_convertible_v<U, T>) optional(U&& u) noexcept
        : value_(std::addressof(u)) {
        static_assert(
            std::is_constructible_v<std::add_lvalue_reference_t<T>, U>,
            "Must be able to bind U to T&");
        static_assert(std::is_lvalue_reference<U>::value,
                      "U must be an lvalue");
    }

    template <class U>
    constexpr explicit(!std::is_convertible_v<U, T>)
        optional(const optional<U>& rhs) noexcept
        : optional(*rhs) {}

    //  \rSec3[optional.dtor]{Destructor}

    constexpr ~optional() = default;

    // \rSec3[optional.assign]{Assignment}
```

```
constexpr optional& operator=(nullopt_t) noexcept {
    value_ = nullptr;
    return *this;
}

constexpr optional& operator=(const optional& rhs) noexcept = default;
constexpr optional& operator=(optional&& rhs) noexcept      = default;

template <class U = T>
    requires(!detail::is_optional<std::decay_t<U>>::value)
constexpr optional& operator=(U&& u) {
    static_assert(
        std::is_constructible_v<std::add_lvalue_reference_t<T>, U>,
        "Must be able to bind U to T&");
    static_assert(std::is_lvalue_reference<U>::value,
                  "U must be an lvalue");
    value_ = std::addressof(u);
    return *this;
}

template <class U>
constexpr optional& operator=(const optional<U>& rhs) noexcept {
    static_assert(
        std::is_constructible_v<std::add_lvalue_reference_t<T>, U>,
        "Must be able to bind U to T&");
    value_ = std::addressof(rhs.value());
    return *this;
}

template <class U>
    requires(!detail::is_optional<std::decay_t<U>>::value)
constexpr optional& emplace(U&& u) noexcept {
    return *this = std::forward<U>(u);
}

//   \rSec3[optional.swap]{Swap}

constexpr void swap(optional& rhs) noexcept {
    std::swap(value_, rhs.value_);
}

// \rSec3[optional.observe]{Observers}
constexpr T* operator->() const noexcept { return value_; }

constexpr T& operator*() const noexcept { return *value_; }

constexpr explicit operator bool() const noexcept {
    return value_ != nullptr;
}
constexpr bool has_value() const noexcept { return value_ != nullptr; }

constexpr T& value() const {
    if (has_value())
        return *value_;
    throw bad_optional_access();
}

template <class U>
```

```
constexpr T value_or(U&& u) const {
    static_assert(std::is_constructible_v<std::add_lvalue_reference_t<T>,
                                          decltype(u)>,
                  "Must be able to bind u to T&");
    return has_value() ? *value_ : std::forward<U>(u);
}

//   \rSec3[optional.monadic]{Monadic operations}

template <class F>
constexpr auto and_then(F&& f) const {
    using U = std::invoke_result_t<F, T&>;
    static_assert(detail::is_optional<U>::value,
                  "F must return an optional");
    return (has_value()) ? std::invoke(std::forward<F>(f), *value_)
                         : std::remove_cvref_t<U>();
}

template <class F>
constexpr auto
transform(F&& f) const -> optional<std::invoke_result_t<F, T&>> {
    using U = std::invoke_result_t<F, T&>;
    return (has_value())
               ? optional<U>{std::invoke(std::forward<F>(f), *value_)}
               : optional<U>{};
}

template <class F>
constexpr optional or_else(F&& f) const {
    using U = std::invoke_result_t<F>;
    static_assert(std::is_same_v<std::remove_cvref_t<U>, optional>);
    return has_value() ? *value_ : std::forward<F>(f)();
}

constexpr void reset() noexcept { value_ = nullptr; }
using iterator       = T*;
using const_iterator = const T*;

// [optional.iterators], iterator support
constexpr T* begin() noexcept {
    if (has_value()) {
        return value_;
    } else {
        return nullptr;
    }
}
constexpr const T* begin() const noexcept {
    if (has_value()) {
        return value_;
    } else {
        return nullptr;
    }
}
constexpr T* end() noexcept { return begin() + has_value(); }

constexpr const T* end() const noexcept { return begin() + has_value(); }

constexpr std::reverse_iterator<T*> rbegin() noexcept {
    return reverse_iterator(end());
```

```
    }
    constexpr std::reverse_iterator<const T*> rbegin() const noexcept {
        return reverse_iterator(end());
    }
    constexpr std::reverse_iterator<T*> rend() noexcept {
        return reverse_iterator(begin());
    }
    constexpr std::reverse_iterator<const T*> rend() const noexcept {
        return reverse_iterator(begin());
    }

    constexpr const T* cbegin() const noexcept { return begin(); }
    constexpr const T* cend() const noexcept { return end(); }
    constexpr std::reverse_iterator<const T*> crbegin() const noexcept {
        return rbegin();
    }
    constexpr std::reverse_iterator<const T*> crend() const noexcept {
        return rend();
    }
};
```