std::optional<T&>

Document #:	P2988R0
Date:	2023-12-29
Project:	Programming Language C++
Audience:	LEWG

Reply-to:	Steve Downey <sdowney@gmail.com, sdowney2@bloomberg.net> Peter Sommerlad <peter.cpp@sommerlad.ch>

1 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&`.

2 Comparison table

There are many situations where an optional holding a reference can come in handy. Here we first look at three possible alternative design options for an object retrieval function that might fail to find a corresponding object in a container. Then there are two more examples showing the inferiority of potential workarounds to the missing `std::optional<T&>`.

2.1 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.

Before	After
`Cat* cat = find_cat("Fido");` `if (cat!=nullptr) { return doit(*cat); }`	`std::optional<Cat&> cat = find_cat("Fido");` `return cat.and_then(doit);`

2.2 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.

Before	After

`// use a smart pointer instead of optional<T&>`
`std::shared_ptr<Cat> cat = find_cat("Fido");`
`if (cat != nullptr) {...`
	`std::optional<Cat&> cat = find_cat("Fido");`
	`cat.and_then([](Cat& thecat){...`

2.3 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.

Before	After
`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){...`

2.4 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.

Before	After

`//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);`
	`  });`

2.5 Using `optional<reference_wrapper<T>>`

While `reference_wrapper<T>` implicitly coverts to `T&` in many practical situation, especially in generic code, such an implicit conversion is not triggered, thus requiring `o.value().get()` train wrecks, to access the wrapped reference, when the optional is engaged. In addition it lacks the possible optimization of the internal representation of `optional<T&>`.

Before	After

`// use an optional reference_wrapper instead of`
`std::optional<std::reference_wrapper<Cat>> cat`
`if (cat) {`
`  cat.value().get() =  Cat("Fynn", color::ora`
`...`
	`std::optional<Cat&> cat = find_cat("Fido");`
	`cat.and_then([](Cat& thecat){`
	`  thecat = Cat("Fynn", color::orange);`
	`...`

3 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* [BindRef].

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.

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.

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

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

6.1 Class template optional [optional.optional_ref]

6.1.1 General [optional.optional_ref.general]

```
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 const T& value() const &;                    // freestanding-deleted
        constexpr T& value() &;                                // freestanding-deleted
        constexpr T&& value() &&;                              // freestanding-deleted
        constexpr const T&& value() const &&;                  // freestanding-deleted
        template <class U>  constexpr T value_or(U&&) const&;
        template <class U>  constexpr T value_or(U&&) &&;

    // [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
};
```

6.1.1.1 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&&);
```

6 *Constraints*:

(6.1) `!is-optional<decay_t<U>>::value is true`

7 *Mandates*:

(7.1) – `std::is_constructible_v<std::add_lvalue_reference_t<T>, U>`;

(7.1) – `std::is_lvalue_reference_v<U>`

8 *Effects*: Initializes `val` with the address of u

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

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

10 *Constraints*:

(10.1) `!is-optional<decay_t<U>>::value is true`

11 *Mandates*:

(11.1) – `std::is_constructible_v<std::add_lvalue_reference_t<T>, U>`;

(11.1) – `std::is_lvalue_reference<U>::value`

12 *Effects*:

6.1.1.2 Destructor [optional_ref.dtor]

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

13 *Remarks*: The destructor is trivial.

7 References

[BindRef] JeanHeyd Meneide. To Bind and Loose a Reference.
https://thephd.dev/to-bind-and-loose-a-reference-optional

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