std::optional<T&>

Document #:	D0000R0
Date:	2023-09-28
Project:	Programming Language C++
Audience:	None
Reply-to:	Steve Downey <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

```cpp
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

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

Before

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

After

```cpp
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 [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> provides, with much greater usability.

There are well motivated suggestions that perhaps instead of an optional<T&> there should be an optional_ref 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, and invites fatal inconsistency in the standard library. The 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.

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

      [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 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 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&&) &&;

      [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;
```

Constructors[optional_ref.ctor]

constexpr optional() noexcept;

constexpr optional(nullopt_t) noexcept;

Postconditions: *this does not contain a value.

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

constexpr optional(const optional& rhs);

Effects: If rhs contains a value, direct-non-list-initializes the contained value with *rhs.

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

6 References

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