std::optional<T&>

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

Document #:	P2988R8
Date:	2024-10-15
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 R7**
  - Wording mandates/constraint fixes
  - Hash on T& pulled out
  - Notes on wording rendering
  - “Fix” make_optional<T&>
- **Changes since R6**
  - strike refref specialization
  - add converting assignment operator
  - add converting in place constructor
- **Changes since R5**
  - refref specialization

    - fix monadic constraints on base template
- **Changes since R4**
    - feature test macro
    - value_or updates from P3091
- **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 existing 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.~~

With further research, the existing uses of make_optional<X&> seem to be primarily test cases, and deliberate use seems to be exceedingly rare in the wild. Reflector review was much more positive about removing the misleading ability to create an `optional<X>` via `make_optional<X&>(x)`. In addition, the multiple argument forms can be used to attempt to construct a optional that contains a reference, but this becomes ill formed because of existing mandates at the type level. In order to preserve existing behavior, where make_optional is not well formed if it constructs a reference, changes to `make_optional` should be made.

Adding a non-type template parameter as the first template parameter to the single argument `make_optional` and mandating that the multi-argument version not request a reference type as the parameter, will diagnose mistaken use of `make_optional` and preserve the existing behavior.

Since construction of an object in order to make a reference to it to construct an optional containing a reference would always dangle, there do not seem to be any use cases for the multi-argument or initializer list forms of make_optional for reference types, and the constructor form seems to satisfy all cases for single argument construction of a optional containing a reference, there does not seem to be a need for a factory function for optional over reference.

There was also discussion of using `std::reference_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 specialization, 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. Implementations of optionals that support reference semantics diverge over the return type, and the current one is arguably 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 intends 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 [P1255R12]. 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 in_place_t construction

The reference specialization allows a limited form of in_place construction where the argument can be converted to the appropriate reference without creation of a temporary. As the reference specialization is non-owning, there is no "place" for a temporary to be constructed that will not dangle. For cases where the lifetime of the constructed object would match the lifetime of the optional, the temporary can be constructed explicitly, instead.

3.9 Converting assignment

A similarly limited converting assignment operator is provided for cases where an optional<U> has a value or refers to a value which can be converted to a T& without construction of a temporary. In particular, converting an optional<T&> to an optional<T const&> is supported.

3.10 Compiler Explorer Playground

See `https://compiler-explorer.com/z/zKqE3sn87` for an updated playground with relevant Google Test functions and various optional implementations made available for cross reference including a flattened in-place version of the reference implementation.

4 Principles for Reification of Design

Optional must never construct a temporary, or knowingly take the address of an temporary or part of an temporary.

It is always presumed safe to copy the pointer value from an optional, since by induction, it is not dangling.

Optional has no storage, so should never construct a T, it may convert a U to a T, so long as that conversion does not create a temporary.

Constructors that would convert from temporary are marked deleted. They should be sufficiently constrained that it was the correct choice and there is no more general, less constrained constructor that would not have created a dangling pointer.

Failure to compile either by ambiguity or no eligible constructors in the overload set is preferable to optional being responsible for use after free or dangling.

Assignment is always from an optional, which may have been an implicit construction. The assignment cannot throw, the construction/conversion may. The assignment may therefore need annotation converting the rhs if that constructor was explicit. This must not be necessary in the default case of creating an optional reference to an lvalue of the same type.

The model for the constraints and mandates for `optional<T&>` is taken from `std::tuple` over reference types. The type `std::tuple` takes the most care of types in the standard library in dealing with creation of temporaries.

As `optional` is designed to be converting, to create instances from arguments that can be used to create the underlying type, constructors should be explicit only where the operations used to create the pointer or the notional reference would be or are explicit.

4.1 Construction from temporary

We disallow construction of `optional<T&>` from any type U in which:

— the constructor body will create a temporary and bind it to a reference.

— a const lvalue reference would be bound to rvalue.

An example of the first case would be construction `optional<std::string const&>` from `char const*`. These cases always dangle.

An example of the second case would be a construction `std::optional<std::string const&>` from temporary `std::string`.

Prohibiting the second case does prevent some safe uses of the optional as the function parameter.

Given:

```
void process(std::optional<std::string const&> arg);
```

This will make a `process(std::string("sdfd"))` invocation ill-formed, despite the arg being safe to use from within the function body.

This deviates from the design of the "view" parameters type, like `std::string_view` or `std::span`. However, we believe that this is the right choice due to the following:

— Only a subset of cases would be working. As an illustration the very similar `process("text")` invocation is ill-formed, due to always being dangling.

— Such design leads to the detection of reference to temporaries or local variables when `optional<T const&>` is used as the return type.

```
std::optional<std::string const&> getValue() {
  std::string localString;
  return localString; // Ill-formed.
  std::optional<std::string> localOptionalString;
  return localOptionalString; // ill-formed
}
```

One of the main motivational examples of `optional<T const&>` is return from a lookup function, and eliminating dangling in such cases outweighs parameter cases.

We are very grateful to Arthur O'Dwyer for his work on [P2266R3] P2266R3 Simpler implicit move accepted in C++23, which makes it possible to implement this correctly.

— We provide behavior consistent with `reference_wrapper<T const>`, that disallows binding to xvalues. We believe that `reference_wrapper<T>` is closer in spirit to `optional<T&>` than any view type. It certainly shares some of the features.

4.2 Deleting dangling overloads

To achieve the dangling safety expressed before, the constructor is marked deleted if it would lead to binding of the reference to temporary or the xvalue. However, deleted constructors are still considered to be candidates during overload resolution, leading to ambiguity in the following examples:

```
void process(std::optional<std::string const&>);
void process(std::optional<char const* const&>);

void test() {
  char const* cstr = Text;
  std::string s = cstr;
  process(s); // Picks, optional<std::string const&> overload
  process(cstr); // Ambiguous, but only std::optional<char const* const&> is not dangling
}
```

During the reflector discussion, an option of an alternate design was presented, where the dangling overload would be constrained, and eliminated from the overload set.

We strongly oppose changing this behavior, as:

— We think that it is impossible to detect temporary binding to xvalue in such a design.

— The behavior we propose is consistent with the behavior for optional for object types

```
void processVal(std::optional<std::string>);
void processVal(std::optional<char const*>);

void test() {
  char const* cstr = "Text";
  std::string s = cstr;
  processVal(s); // Picks std::string overload
  processVal(cstr); // Ambiguous
}
```

As language in general treats functions accepting by value and by const reference in the same manner during overload resolution, we believe achieving this consistency is a feature.

The design that was introduced by `std::tuple`, and `std::pair`, for references, is followed, where the detection of dangling does not affect the results of overload resolution and instead makes a call that would dangle be ill-formed and diagnosed.

4.3 Assignment of optional<T&>

In the case of `optional<T&>`, any assignment operation is equivalent to assigning a pointer, and there is no observable difference between: using converting assignment from `U&&` or `optional<U>` constructing temporary `optional<T&>`, and then assigning it to it.

This observation allows us to provide only copy-assignment for `optional<T&>`, instead of a set of converting assignments, that would need to replicate the signatures of constructors and their constraints. Assignment from any other value is handled by first implicitly constructing `optional<T&>` and then using copy-assignment. Move-assignment is the same as copy-assignment, since only pointer copy is involved.

5 Proposal

Add an lvalue reference specialization for the std::optional template.

6 Wording

The wording here cross references and adopts the wording in [P3091R2].

�.1 Optional objects [optional]

�.�.1 General [optional.general]

1 Subclause �.1 describes class template `optional` that represents optional objects. An *optional object* is an object that contains the storage for another object and manages the lifetime of this contained object, if any. The contained object may be initialized after the optional object has been initialized, and may be destroyed before the optional object has been destroyed. The initialization state of the contained object is tracked by the optional object.

�.�.2 Header `<optional>` synopsis [optional.syn]

```
// mostly freestanding
#include <compare>              // see ??

namespace std {
  // �.�.3, class template optional for object types
  template <class T>
  class optional; // partially freestanding

  // �.�.4, class template optional for reference types
  template <class T>
  class optional<T&>; // partially freestanding

  template <class T>
  inline constexpr bool ranges::enable_view<optional<T>> = true;
  template <class T>
  inline constexpr auto format_kind<optional<T>> = range_format::disabled;
  template<class T>
  constexpr bool ranges::enable_borrowed_range<optional<T>> = is_reference_v<T>;

  template<class T>
    concept is-derived-from-optional = requires(const T& t) {       // exposition only
      []<class U>(const optional<U>&){ }(t);
    };

  // �.�.5, no-value state indicator
  struct nullopt_t{see below};
  inline constexpr nullopt_t nullopt(unspecified);

  // �.�.6, class bad_optional_access
  class bad_optional_access;

  // �.�.7, relational operators
  template<class T, class U>
    constexpr bool operator==(const optional<T>&, const optional<U>&);
  template<class T, class U>
    constexpr bool operator!=(const optional<T>&, const optional<U>&);
  template<class T, class U>
    constexpr bool operator<(const optional<T>&, const optional<U>&);
  template<class T, class U>
    constexpr bool operator>(const optional<T>&, const optional<U>&);
  template<class T, class U>
    constexpr bool operator<=(const optional<T>&, const optional<U>&);
  template<class T, class U>
    constexpr bool operator>=(const optional<T>&, const optional<U>&);
  template<class T, three_way_comparable_with<T> U>
    constexpr compare_three_way_result_t<T, U>
      operator<=>(const optional<T>&, const optional<U>&);

  // ??, comparison with nullopt
  template<class T> constexpr bool operator==(const optional<T>&, nullopt_t) noexcept;
  template<class T>
    constexpr strong_ordering operator<=>(const optional<T>&, nullopt_t) noexcept;
  // ??, comparison with T
  template<class T, class U> constexpr bool operator==(const optional<T>&, const U&);
  template<class T, class U> constexpr bool operator==(const T&, const optional<U>&);
  template<class T, class U> constexpr bool operator!=(const optional<T>&, const U&);
  template<class T, class U> constexpr bool operator!=(const T&, const optional<U>&);
  template<class T, class U> constexpr bool operator<(const optional<T>&, const U&);
  template<class T, class U> constexpr bool operator<(const T&, const optional<U>&);
  template<class T, class U> constexpr bool operator>(const optional<T>&, const U&);
  template<class T, class U> constexpr bool operator>(const T&, const optional<U>&);
  template<class T, class U> constexpr bool operator<=(const optional<T>&, const U&);
  template<class T, class U> constexpr bool operator<=(const T&, const optional<U>&);
  template<class T, class U> constexpr bool operator>=(const optional<T>&, const U&);
  template<class T, class U> constexpr bool operator>=(const T&, const optional<U>&);
  template<class T, class U>
      requires (!is-derived-from-optional<U>) && three_way_comparable_with<T, U>
    constexpr compare_three_way_result_t<T, U>
      operator<=>(const optional<T>&, const U&);

  // �.�.8, specialized algorithms
  template<class T>
    constexpr void swap(optional<T>&, optional<T>&) noexcept(see below);

  template<class T>
    constexpr optional<see below> make_optional(T&&);
  template<class T, class... Args>
    constexpr optional<T> make_optional(Args&&... args);
  template<class T, class U, class... Args>
    constexpr optional<T> make_optional(initializer_list<U> il, Args&&... args);

  // �.�.9, hash support
  template<class T> struct hash;
  template<class T> struct hash<optional<T>>;
}
```

�.�.3 Class template `optional` for object types [optional.optional]

�.�.3.1 General [optional.optional.general]

```
namespace std {
  template<class T>
  class optional {
  public:
    using value_type     = T;
    using iterator       = implementation-defined;            // see �.�.3.6
    using const_iterator = implementation-defined;            // see �.�.3.6

    // �.�.3.2, constructors
    constexpr optional() noexcept;
    constexpr optional(nullopt_t) noexcept;
    constexpr optional(const optional&);
    constexpr optional(optional&&) noexcept(see below);

    template<class... Args>
      constexpr explicit optional(in_place_t, Args&&...);
    template<class U, class... Args>
      constexpr explicit optional(in_place_t, initializer_list<U>, Args&&...);
    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>&&);
    // �.�.3.3, destructor
    constexpr ~optional();

    // �.�.3.4, 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>&&);
    template<class... Args> constexpr T& emplace(Args&&...);
    template<class U, class... Args> constexpr T& emplace(initializer_list<U>, Args&&...);

    // �.�.3.5, swap
    constexpr void swap(optional&) noexcept(see below);

    // �.�.3.6, iterator support
    constexpr iterator begin() noexcept;
    constexpr const_iterator begin() const noexcept;
    constexpr iterator end() noexcept;
    constexpr const_iterator end() const noexcept;

    // �.�.3.7, 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 &;                                    // 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&&) &&;

    // �.�.3.8, 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 &;

    // �.�.3.9, modifiers
    constexpr void reset() noexcept;

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

  template<class T>
    optional(T) -> optional<T>;
}
```

1 Any instance of `optional<T>` at any given time either contains a value or does not contain a value. When an instance of `optional<T>` *contains a value,* it means that an object of type `T`, referred to as the optional object's

contained value, is allocated within the storage of the optional object. Implementations are not permitted to use additional storage, such as dynamic memory, to allocate its contained value. When an object of type `optional<T>` is contextually converted to `bool`, the conversion returns `true` if the object contains a value; otherwise the conversion returns `false`.

2 When an `optional<T>` object contains a value, member `val` points to the contained value.

~~`T` shall be a type other than *cv* `in_place_t` or *cv* `nullopt_t` that meets the *Cpp17Destructible* requirements (cpp17.destructible).~~

3 A type `X` is a *valid element type* for `optional` if `remove_cv_t<X>` is an lvalue reference type, or a complete non-array object type, and `remove_cvref_t<X>` is a type other than `in_place_t` or `nullopt_t`.

4 `T` shall be an object type that is valid element type for `optional` and meets the *Cpp17Destructible* requirements (cpp17.destructible).

�.�.3.2 Constructors [optional.ctor]

�.�.3.3 Destructor [optional.dtor]

�.�.3.4 Assignment [optional.assign]

�.�.3.5 Swap [optional.swap]

�.�.3.6 Iterator support [optional.iterators]

�.�.3.7 Observers [optional.observe]

�.�.3.8 Monadic operations [optional.monadic]

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

1 Let `U` be `invoke_result_t<F, decltype(*val)>`.

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

3 *Effects*: Equivalent to:

```
if (*this) {
  return invoke(std::forward<F>(f), *val);
} else {
  return remove_cvref_t<U>();
}
```

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

4 Let `U` be `invoke_result_t<F, decltype(std::move(*val))>`.

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

6 *Effects*: Equivalent to:

```
if (*this) {
  return invoke(std::forward<F>(f), std::move(*val));
} else {
  return remove_cvref_t<U>();
}
```

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

7 Let `U` be `remove_cv_t<invoke_result_t<F, decltype(*val)>>`.

8 *Mandates*: ~~`U` is a non-array object type other than `in_place_t` or `nullopt_t`.~~ `U` is valid element type for optional. The declaration

```
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*]

9 *Returns*: If `*this` contains a value, an `optional<U>` object whose contained value is direct-non-list-initialized with `invoke(std::forward<F>(f), *val)`; otherwise, `optional<U>()`.

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

10 Let `U` be `remove_cv_t<invoke_result_t<F, decltype(std::move(*val))>>`.

11 *Mandates*: `U` is a non-array object type other than `in_place_t` or `nullopt_t`. `U` is valid element type for optional. The declaration

```
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*]

12 *Returns*: If `*this` contains a value, an `optional<U>` object whose contained value is direct-non-list-initialized with `invoke(std::forward<F>(f), std::move(*val))`; otherwise, `optional<U>()`.

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

13 *Constraints*: `F` models `invocable<>` and `T` models `copy_constructible`.

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

15 *Effects*: Equivalent to:

```
if (*this) {
  return *this;
} else {
  return std::forward<F>(f)();
}
```

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

16 *Constraints*: `F` models `invocable<>` and `T` models `move_constructible`.

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

18 *Effects*: Equivalent to:

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

�.�.3.9 Modifiers [optional.mod]

�.�.4 Class template `optional` for reference types [optional.optionalref]

�.�.4.1 General [optional.optionalref.general]

```
namespace std {
  template<class T>
  class optional<T&> {
    public:
      using value_type     = T;
      using iterator       = implementation-defined; // see �.�.4.5

    public:
      // �.�.4.2, constructors
      constexpr optional() noexcept;
      constexpr optional(nullopt_t) noexcept;
      constexpr optional(const optional& rhs) noexcept = default;

      template <class Arg>
      constexpr explicit optional(in_place_t, Arg&& arg);

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

      constexpr ~optional() = default;

      // �.�.4.3, assignment
      constexpr optional& operator=(nullopt_t) noexcept;
      constexpr optional& operator=(const optional& rhs) noexcept = default;

      template <class U> constexpr optional& emplace(U&& u) noexcept(see below);

      // �.�.4.4, swap
      constexpr void swap(optional& rhs) noexcept;

      // �.�.3.6, iterator support
      constexpr iterator begin() const noexcept;
      constexpr iterator end() const noexcept;

      // �.�.4.6, observers
      constexpr T*       operator->() const noexcept;
      constexpr T&       operator*() const noexcept;
      constexpr explicit operator bool() const noexcept;
      constexpr bool     has_value() const noexcept;
      constexpr T&       value() const;                              // freestanding-deleted
      template <class U> constexpr remove_cv_t<T> value_or(U&& u) const;

      // �.�.4.7, monadic operations
      template <class F> constexpr auto and_then(F&& f) const;
      template <class F> constexpr auto transform(F&& f) const -> optional<invoke_result_t<F, T&>>;
      template <class F> constexpr optional or_else(F&& f) const;

      // �.�.4.8, modifiers
      constexpr void reset() noexcept;

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

}
```

1 An instance of `optional<T&>` contains a value if *`val`* `!= nullptr` is `true`.

�.�.4.2 Constructors [optionalref.ctor]

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

1 *Effects*: Equivalent to

```
  val = nullptr;
```

```
template <class Arg>
constexpr explicit optional(in_place_t, Arg&& arg);
```

2 *Constraints*:

(2.1) — `is_constructible_v<T&, remove_cv_t<Arg>>` is `true`

(2.2) — `reference_constructs_from_temporary_v<T&, Arg>)` is `false`

3 *Effects*: Initializes *`val`* with `addressof(static_cast<T&>(std::forward<Arg>(arg)))`

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

4 *Constraints*:

(4.1) — `is_constructible_v<T&, U>` is `true`

(4.2) — `(is_same_v<remove_cvref_t<U>, in_place_t>)` is `false`

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

5 *Effects*: Initializes *`val`* with `addressof(static_cast<T&>(std::forward<U>(u)))`.

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

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

The expression inside `noexcept` is equivalent to:

```
is_nothrow_constructible_v<T&, U>
```

This constructor is defined as deleted if

```
reference_constructs_from_temporary_v<T&, U>
```

is `true`

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

7 *Constraints*:

(7.1) — `is_constructible_v<T&, U&>` is `true`

(7.2) — `std::is_same_v<remove_cv_t<T>, optional<U>>` is `false`

(7.3) — `std::is_same_v<T&, U>` is `false`

8 *Effects*: Equivalent to

```
if (rhs.has_value()) {
    val = addressof(static_cast<T&>(rhs.value()));
} else {
    val = nullptr;
}
```

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

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

The expression inside `noexcept` is equivalent to:

```
is_nothrow_constructible_v<T&, U&>
```

This constructor is defined as deleted if

```
reference_constructs_from_temporary_v<T&, U&>
```

is `true`

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

10 *Constraints*:

(10.1) — `is_constructible_v<T&, const U&>` is `true`

(10.2) — `std::is_same_v<remove_cv_t<T>, optional<U>>` is `false`

(10.3) — `std::is_same_v<T&, U>` is `false`

11 *Effects*: Equivalent to

```
if (rhs.has_value()) {
    val = addressof(static_cast<T&>(rhs.value()));
} else {
    val = nullptr;
}
```

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

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

The expression inside `noexcept` is equivalent to:

```
is_nothrow_constructible_v<T&, const U&>
```

This constructor is defined as deleted if

```
reference_constructs_from_temporary_v<T&, const U&>
```

is `true`

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

13 *Constraints*:

(13.1) — `is_constructible_v<T&, U>` is `true`

(13.2) — `std::is_same_v<remove_cv_t<T>, optional<U>>` is `false`

(13.3) — `std::is_same_v<T&, U>` is `false`

14 *Effects*: Equivalent to

```
if (rhs.has_value()) {
    val = addressof(static_cast<T&>(std::move(rhs).value()));
} else {
    val = nullptr;
}
```

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

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

The expression inside `noexcept` is equivalent to:

```
is_nothrow_constructible_v<T&, U>
```

This constructor is defined as deleted if

```
reference_constructs_from_temporary_v<T&, U>
```

is `true`

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

16 *Constraints*:

(16.1) — `is_constructible_v<T&, const U>` is `true`

(16.2) — `std::is_same_v<remove_cv_t<T>, optional<U>>` is `false`

(16.3) — `std::is_same_v<T&, U>` is `false`

17 *Effects*: Equivalent to

```
if (rhs.has_value()) {
    val = addressof(static_cast<T&>(std::move(rhs).value()));
} else {
    val = nullptr;
}
```

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

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

The expression inside `noexcept` is equivalent to:

```
is_nothrow_constructible_v<T&, const U>
```

This constructor is defined as deleted if

```
reference_constructs_from_temporary_v<T&, const U>
```

is `true`

�.�.4.3 Assignment [optionalref.assign]

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

1 *Effects*: Equivalent to

```
val = nullptr;
```

2 *Returns*: `*this`.

```
template <class U>
constexpr T& emplace(U&& u) noexcept(see below);
```

3 *Constraints*:

(3.1) — `is_constructible_v<T&, U>` is `true`

(3.2) — `reference_constructs_from_temporary_v<T&, U>` is `false`

4 *Effects*: Assigns *`val`* `addressof(static_cast<T&>(std::forward<U>(u)))`

5 *Remarks*: The expression inside `noexcept` is equivalent to:

```
is_nothrow_constructible_v<T&, U>
```

�.�.4.4 Swap [optionalref.swap]

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

1 *Effects*: Equivalent to

```
std::swap(val, rhs.val);
```

�.�.4.5 Iterator support [optionalref.iterators]

```
using iterator = implementation-defined;
```

1 These types model `contiguous_iterator`, meet the *Cpp17RandomAccessIterator* requirements, and meet the requirements for constexpr iterators, with value type `remove_cv_t<T>`. The reference type is `T&` for `iterator`.

2 All requirements on container iterators apply to `optional::iterator`.

```
constexpr iterator begin() const noexcept;
```

3 *Returns*: If `has_value()` is `true`, an iterator referring to the referred to value. Otherwise, a past-the-end iterator value.

```
constexpr iterator end() const noexcept;
```

4 *Returns*: `begin() + has_value()`.

�.�.4.6 Observers [optionalref.observe]

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

1 *Returns*: *`val`*.

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

2 *Preconditions*: `has_value()` is `true`

3 *Returns*: `*`*`val`*.

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

4 *Returns*: *`val`* `!= nullptr`

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

5 *Returns*: *`val`* `!= nullptr`

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

6 *Effects*: Equivalent to:

```
if (has_value()) {
  return *val;
}
throw bad_optional_access();
```

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

7 *Mandates*:

(7.1) — `is_constructible_v<std::remove_cv_t<T>, T&>` is `true`.

(7.2) — `is_convertible_v<U, std::remove_cv_t<T>>` is `true`.

8 *Effects*: Equivalent to:

```
if (has_value()) {
    return remove_cv_t<T>(*val);
} else {
    return std::forward<U>(u);
}
```

�.�.4.7 Monadic operations [optionalref.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:

```
if (has_value()) {
    return invoke(std::forward<F>(f), *val);
} else {
    return remove_cvref_t<U>();
}
```

```
template <class F>
constexpr auto transform(F&& f) const -> optional<invoke_result_t<F, T&>>;
```

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

5 *Mandates*: `U` is a valid element type for optional.

6 *Effects*: Equivalent to:

```
if (has_value()) {
    return optional<U>{invoke(std::forward<F>(f), *val)};
} else {
    return optional<U>{};
}
```

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

7 *Mandates*: `is_same_v<remove_cvref_t<U>, optional>` is `true`.

8 *Effects*: Equivalent to:

```
if (has_value()) {
    return *val;
} else {
    return std::forward<F>(f)();
}
```

�.�.4.8 Modifiers [optionalref.mod]

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

1 *Effects*: Equivalent to

```
val = nullptr;
```

val does not refer to a value.

�.�.5 No-value state indicator [optional.nullopt]

�.�.6 Class `bad_optional_access` [optional.bad.access]

�.�.7 Relational operators [optional.relops]

�.�.8 Specialized algorithms [optional.specalg]

```
template<class T>
  constexpr void swap(optional<T>& x, optional<T>& y) noexcept(noexcept(x.swap(y)));
```

1 *Constraints*: `is_move_constructible_v<T>` is `true` and `is_swappable_v<T>` is `true`.

Constraints: `is_reference_v<T> || (is_move_constructible_v<T> && is_swappable_v<T>)` is `true`.

2 *Effects*: Calls `x.swap(y)`.

```
template<class T> constexpr optional<decay_t<T>> make_optional(T&& v);
```

3 *Constraints*: The call to `make_optional` does not use an explicit *template-argument-list* that begins with a type *template-argument*.

4 *Returns*: `optional<decay_t<T>>(std::forward<T>(v))`.

```
template<class T, class...Args>
  constexpr optional<T> make_optional(Args&&... args);
```

5 *Mandates*: `is_reference_v<T>` is `false`.

6 *Effects*: Equivalent to: `return optional<T>(in_place, std::forward<Args>(args)...);`

```
template<class T, class U, class... Args>
  constexpr optional<T> make_optional(initializer_list<U> il, Args&&... args);
```

7 *Mandates*: `is_reference_v<U>` is `false`.

8 *Effects*: Equivalent to: `return optional<T>(in_place, il, std::forward<Args>(args)...);`

�.�.9 Hash support [optional.hash]

`template<class T> struct hash<optional<T>>;`

1 The specialization `hash<optional<T>>` is enabled if and only if is_reference_v<T> is false and `hash<remove_-const_t<T>>` is enabled. When enabled, for an object `o` of type `optional<T>`, if `o.has_value() == true`, then `hash<optional<T>>()(o)` evaluates to the same value as `hash<remove_const_t<T>>()(*o)`; otherwise it evaluates to an unspecified value. The member functions are not guaranteed to be `noexcept`.

�.�.10 Feature-test macro [version.syn]

Add the following macro definition to [version.syn], header <version> synopsis, with the value selected by the editor to reflect the date of adoption of this paper:

```
#define __cpp_lib_optional_ref 20XXXXL // also in <ranges>, <tuple>, <utility>
```

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 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/Optional26`.

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

[P1255R12] Steve Downey. P1255R12: A view of 0 or 1 elements: views::maybe. `https://wg21.link/p1255r12`, 1 2024.

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

[P2266R3] Arthur O'Dwyer. P2266R3: Simpler implicit move. `https://wg21.link/p2266r3`, 3 2022.

[P3091R2] Pablo Halpern. P3091R2: Better lookups for 'map' and 'unordered_map'. `https://wg21.link/p3091r2`, 5 2024.

[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. optional.hpp. `https://raw.githubusercontent.com/steve-downey/Optional26/refref/include/Beman/Optional26/optional.hpp`. (Accessed on 08/14/2024).

Implementation

```
// ----------------------
// BASE AND DETAILS ELIDED
// ----------------------

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

template <class T>
class optional<T&> {
  public:
  using value_type = T&;
  using iterator =
  detail::contiguous_iterator<T,
  optional>; // see [optionalref.iterators]
  using const_iterator =
  detail::contiguous_iterator<const T,
  optional>; // see [optionalref.iterators]

  private:
  template <class R, class Arg>
  constexpr R make_reference(Arg&& arg) // exposition only
  requires is_constructible_v<R, Arg>;

  public:
  // \ref{optionalref.ctor}, constructors

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

  template <class Arg>
  constexpr explicit optional(in_place_t, Arg&& arg)
  requires is_constructible_v<add_lvalue_reference_t<T>, Arg>;

  template <class U = T>
  requires(!detail::is_optional<decay_t<U>>)
  constexpr explicit(!is_convertible_v<U, T>) optional(U&& u) noexcept;
  template <class U>
  constexpr explicit(!is_convertible_v<U, T>)
  optional(const optional<U>& rhs) noexcept;

  // \ref{optionalref.dtor}, destructor
  constexpr ~optional() = default;

  // \ref{optionalref.assign}, assignment
  constexpr optional& operator=(nullopt_t) noexcept;

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

  template <class U = T>
  requires(!detail::is_optional<decay_t<U>>)
  constexpr optional& operator=(U&& u);
  template <class U>
  constexpr optional& operator=(const optional<U>& rhs) noexcept;

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

  template <class U>
  requires(!detail::is_optional<decay_t<U>>)
  constexpr optional& emplace(U&& u) noexcept;

  // \ref{optionalref.swap}, swap
  constexpr void swap(optional& rhs) noexcept;

  // \ref{optional.iterators}, iterator support
  constexpr iterator       begin() noexcept;
  constexpr const_iterator begin() const noexcept;
  constexpr iterator       end() noexcept;
  constexpr const_iterator end() const noexcept;

  // \ref{optionalref.observe}, observers
  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;
  template <class U>
  constexpr T value_or(U&& u) const;

  // \ref{optionalref.monadic}, monadic operations
  template <class F>
  constexpr auto and_then(F&& f) const;
  template <class F>
  constexpr auto transform(F&& f) const -> optional<invoke_result_t<F, T&>>;
  template <class F>
  constexpr optional or_else(F&& f) const;

  // \ref{optional.mod}, modifiers
  constexpr void reset() noexcept;

  private:
  T* value_; // exposition only
};

template <class T>
template <class R, class Arg>
constexpr R optional<T&>::make_reference(Arg&& arg)
requires is_constructible_v<R, Arg>
{
  static_assert(std::is_reference_v<R>);
  #if (__cpp_lib_reference_from_temporary >= 202202L)
  static_assert(!std::reference_converts_from_temporary_v<R, Arg>,
  "Reference conversion from temporary not allowed.");
  #endif
  R r(std::forward<Arg>(arg));
  return r;
}

//  \rSec3[optionalref.ctor]{Constructors}
template <class T>
```

```
constexpr optional<T&>::optional() noexcept : value_(nullptr) {}

template <class T>
constexpr optional<T&>::optional(nullopt_t) noexcept : value_(nullptr) {}

template <class T>
template <class Arg>
constexpr optional<T&>::optional(in_place_t, Arg&& arg)
requires is_constructible_v<add_lvalue_reference_t<T>, Arg>
: value_(addressof(
make_reference<add_lvalue_reference_t<T>>(std::forward<Arg>(arg)))) {
}

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

template <class T>
template <class U>
constexpr optional<T&>::optional(const optional<U>& rhs) noexcept {
  static_assert(is_constructible_v<add_lvalue_reference_t<T>, U>,
  "Must be able to bind U to T&");
  if (rhs.has_value())
  value_ = to_address(rhs);
  else
  value_ = nullptr;
}

// \rSec3[optionalref.assign]{Assignment}
template <class T>
constexpr optional<T&>& optional<T&>::operator=(nullopt_t) noexcept {
  value_ = nullptr;
  return *this;
}

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

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

template <class T>
template <class U>
constexpr optional<T&>& optional<T&>::operator=(optional<U>&& rhs) {
  static_assert(is_constructible_v<add_lvalue_reference_t<T>, U>,
  "Must be able to bind U to T&");
  #if (__cpp_lib_reference_from_temporary >= 202202L)
  static_assert(
  !std::reference_converts_from_temporary_v<add_lvalue_reference_t<T>,
  U>,
  "Reference conversion from temporary not allowed.");
  #endif
  if (rhs.has_value())
  value_ = to_address(rhs);
  else
  value_ = nullptr;
  return *this;
}

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

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

template <class T>
constexpr void optional<T&>::swap(optional<T&>& rhs) noexcept {
  std::swap(value_, rhs.value_);
}

// \rSec3[optionalref.iterators]{Iterator Support}
template <class T>
constexpr optional<T&>::iterator optional<T&>::begin() noexcept {
  return iterator(has_value() ? value_ : nullptr);
};

template <class T>
constexpr optional<T&>::const_iterator optional<T&>::begin() const noexcept {
  return const_iterator(has_value() ? value_ : nullptr);
};

template <class T>
constexpr optional<T&>::iterator optional<T&>::end() noexcept {
  return begin() + has_value();
}

template <class T>
constexpr optional<T&>::const_iterator optional<T&>::end() const noexcept {
  return begin() + has_value();
}

// \rSec3[optionalref.observe]{Observers}
```

```
template <class T>
constexpr T* optional<T&>::operator->() const noexcept {
  return value_;
}

template <class T>
constexpr T& optional<T&>::operator*() const noexcept {
  return *value_;
}

template <class T>
constexpr optional<T&>::operator bool() const noexcept {
  return value_ != nullptr;
}
template <class T>
constexpr bool optional<T&>::has_value() const noexcept {
  return value_ != nullptr;
}

template <class T>
constexpr T& optional<T&>::value() const {
  if (has_value())
  return *value_;
  throw bad_optional_access();
}

template <class T>
template <class U>
constexpr T optional<T&>::value_or(U&& u) const {
  static_assert(is_copy_constructible_v<T>, "T must be copy constructible");
  static_assert(is_convertible_v<decltype(u), T>,
  "Must be able to convert u to T");
  return has_value() ? *value_ : std::forward<U>(u);
}

//   \rSec3[optionalref.monadic]{Monadic operations}
template <class T>
template <class F>
constexpr auto optional<T&>::and_then(F&& f) const {
  using U = invoke_result_t<F, T&>;
  static_assert(detail::is_optional<U>, "F must return an optional");
  return (has_value()) ? invoke(std::forward<F>(f), *value_)
  : remove_cvref_t<U>();
}

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

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

// \rSec3[optional.mod]{modifiers}
template <class T>
constexpr void optional<T&>::reset() noexcept {
  value_ = nullptr;
}
```