std::optional<T&>

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

Document #:	P2988R6
Date:	2024-08-15
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 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.

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 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 lvalue and rvalue reference specializations for the std::optional template.

5 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

  // �.�.5, class template optional for rvalue 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>
    concept is-derived-from-optional = requires(const T& t) {       // exposition only
      []<class U>(const optional<U>&){ }(t);
    };

  // �.�.6, no-value state indicator
  struct nullopt_t{see below};
  inline constexpr nullopt_t nullopt(unspecified);

  // �.�.7, class bad_optional_access
  class bad_optional_access;

  // �.�.8, 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>&);
```

```
  // �.�.9, 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;

  // �.�.10, 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&);

  // �.�.11, 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);

  // �.�.12, 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.

3 T shall be a non-array object type other than *cv* in_place_t or *cv* nullopt_t that meets the *Cpp17Destructible* requirements ().

�.�.3.2 Constructors [optional.ctor]

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

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

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

2 *Postconditions*: *this does not contain a value.

3 *Remarks*: No contained value is initialized. For every object type T these constructors are constexpr constructors.

```
constexpr optional(const optional& rhs);
```

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

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

6 *Throws*: Any exception thrown by the selected constructor of T.

7 *Remarks*: This constructor is defined as deleted unless is_copy_constructible_v<T> is true. If is_trivially_copy_constructible_v<T> is true, this constructor is trivial.

```
constexpr optional(optional&& rhs) noexcept(see below);
```

8 *Constraints*: is_move_constructible_v<T> is true.

9 *Effects*: If rhs contains a value, direct-non-list-initializes the contained value with std::move(*rhs). rhs.has_value() is unchanged.

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

11 *Throws*: Any exception thrown by the selected constructor of T.

12 *Remarks*: The exception specification is equivalent to is_nothrow_move_constructible_v<T>. If is_trivially_move_constructible_v<T> is true, this constructor is trivial.

```
template<class... Args> constexpr explicit optional(in_place_t, Args&&... args);
```

13 *Constraints*: is_constructible_v<T, Args...> is true.

14 *Effects*: Direct-non-list-initializes the contained value with std::forward<Args>(args)....

15 *Postconditions*: *this contains a value.

16 *Throws*: Any exception thrown by the selected constructor of T.

17 *Remarks*: If T's constructor selected for the initialization is a constexpr constructor, this constructor is a constexpr constructor.

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

18 *Constraints*: is_constructible_v<T, initializer_list<U>&, Args...> is true.

19 *Effects*: Direct-non-list-initializes the contained value with il, std::forward<Args>(args)....

20 *Postconditions*: *this contains a value.

21 *Throws*: Any exception thrown by the selected constructor of T.

22 *Remarks*: If T's constructor selected for the initialization is a constexpr constructor, this constructor is a constexpr constructor.

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

23 *Constraints*:

(23.1) — is_constructible_v<T, U> is true,

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

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

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

24 *Effects*: Direct-non-list-initializes the contained value with std::forward<U>(v).

25 *Postconditions*: *this contains a value.

26 *Throws*: Any exception thrown by the selected constructor of T.

27 *Remarks*: If T's selected constructor is a constexpr constructor, this constructor is a constexpr constructor. The expression inside explicit is equivalent to:

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

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

28 *Constraints*:

(28.1) — is_constructible_v<T, const U&> is true, and

(28.2) — if T is not *cv* bool, *converts-from-any-cvref*<T, optional<U>> is false.

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

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

31 *Throws*: Any exception thrown by the selected constructor of T.

32 *Remarks*: The expression inside explicit is equivalent to:

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

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

33 *Constraints*:

(33.1) — is_constructible_v<T, U> is true, and

(33.2) — if T is not *cv* bool, *converts-from-any-cvref*<T, optional<U>> is false.

34 *Effects*: If rhs contains a value, direct-non-list-initializes the contained value with std::move(*rhs). rhs.has_value() is unchanged.

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

36 *Throws*: Any exception thrown by the selected constructor of T.

37 *Remarks*: The expression inside explicit is equivalent to:

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

�.�.3.3 Destructor [optional.dtor]

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

1 *Effects*: If is_trivially_destructible_v<T> != true and *this contains a value, calls

```
val->T::~T()
```

2 *Remarks*: If `is_trivially_destructible_v<T>` is `true`, then this destructor is trivial.

�.�.3.4 Assignment [optional.assign]

```
constexpr optional<T>& operator=(nullopt_t) noexcept;
```

1 *Effects*: If `*this` contains a value, calls `val->T::~T()` to destroy the contained value; otherwise no effect.

2 *Postconditions*: `*this` does not contain a value.

3 *Returns*: `*this`.

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

4 *Effects*: See .

Table 1: `optional::operator=(const optional&)` effects [tab:optional.assign.copy]

	`*this` contains a value	**`*this` does not contain a value**
`rhs` contains a value	assigns `*rhs` to the contained value	direct-non-list-initializes the contained value with `*rhs`
`rhs` does not contain a value	destroys the contained value by calling `val->T::~T()`	no effect

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

6 *Returns*: `*this`.

7 *Remarks*: If any exception is thrown, the result of the expression `this->has_value()` remains unchanged. If an exception is thrown during the call to `T`'s copy constructor, no effect. If an exception is thrown during the call to `T`'s copy assignment, the state of its contained value is as defined by the exception safety guarantee of `T`'s copy assignment. This operator is defined as deleted unless `is_copy_constructible_v<T>` is `true` and `is_copy_assignable_v<T>` is `true`. If `is_trivially_copy_constructible_v<T> && is_trivially_copy_assignable_v<T> && is_trivially_destructible_v<T>` is `true`, this assignment operator is trivial.

```
constexpr optional& operator=(optional&& rhs) noexcept(see below);
```

8 *Constraints*: `is_move_constructible_v<T>` is `true` and `is_move_assignable_v<T>` is `true`.

9 *Effects*: See . The result of the expression `rhs.has_value()` remains unchanged.

Table 2: `optional::operator=(optional&&)` effects [tab:optional.assign.move]

	`*this` contains a value	**`*this` does not contain a value**
`rhs` contains a value	assigns `std::move(*rhs)` to the contained value	direct-non-list-initializes the contained value with `std::move(*rhs)`
`rhs` does not contain a value	destroys the contained value by calling `val->T::~T()`	no effect

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

11 *Returns*: `*this`.

12 *Remarks*: The exception specification is equivalent to:

```
is_nothrow_move_assignable_v<T> && is_nothrow_move_constructible_v<T>
```

13 If any exception is thrown, the result of the expression `this->has_value()` remains unchanged. If an exception is thrown during the call to `T`'s move constructor, the state of `*rhs.val` is determined by the exception safety guarantee of `T`'s move constructor. If an exception is thrown during the call to `T`'s move assignment, the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `T`'s move

assignment. If is_trivially_move_constructible_v<T> && is_trivially_move_assignable_v<T> && is_trivially_destructible_v<T> is true, this assignment operator is trivial.

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

14 *Constraints*: is_same_v<remove_cvref_t<U>, optional> is false, conjunction_v<is_scalar<T>, is_same<T, decay_t<U>>> is false, is_constructible_v<T, U> is true, and is_assignable_v<T&, U> is true.

15 *Effects*: If *this contains a value, assigns std::forward<U>(v) to the contained value; otherwise direct-non-list-initializes the contained value with std::forward<U>(v).

16 *Postconditions*: *this contains a value.

17 *Returns*: *this.

18 *Remarks*: If any exception is thrown, the result of the expression this->has_value() remains unchanged. If an exception is thrown during the call to T's constructor, the state of v is determined by the exception safety guarantee of T's constructor. If an exception is thrown during the call to T's assignment, the state of *val and v is determined by the exception safety guarantee of T's assignment.

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

19 *Constraints*:

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

(19.2) — is_assignable_v<T&, const U&> is true,

(19.3) — *converts-from-any-cvref*<T, optional<U>> is false,

(19.4) — is_assignable_v<T&, optional<U>&> is false,

(19.5) — is_assignable_v<T&, optional<U>&&> is false,

(19.6) — is_assignable_v<T&, const optional<U>&> is false, and

(19.7) — is_assignable_v<T&, const optional<U>&&> is false.

20 *Effects*: See .

Table 3: optional::operator=(const optional<U>&) effects [tab:optional.assign.copy.templ]

	***this contains a value**	***this does not contain a value**
rhs contains a value	assigns *rhs to the contained value	direct-non-list-initializes the contained value with *rhs
rhs does not contain a value	destroys the contained value by calling val->T::~T()	no effect

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

22 *Returns*: *this.

23 *Remarks*: If any exception is thrown, the result of the expression this->has_value() remains unchanged. If an exception is thrown during the call to T's constructor, the state of *rhs.val is determined by the exception safety guarantee of T's constructor. If an exception is thrown during the call to T's assignment, the state of *val and *rhs.val is determined by the exception safety guarantee of T's assignment.

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

24 *Constraints*:

(24.1) — is_constructible_v<T, U> is true,

(24.2) — is_assignable_v<T&, U> is true,

(24.3) — *converts-from-any-cvref*<T, optional<U>> is false,

(24.4) — is_assignable_v<T&, optional<U>&> is false,

(24.5) — is_assignable_v<T&, optional<U>&&> is false, (24.6) — `is_assignable_v<T&, const optional<U>&>` is `false`, and

(24.7) — `is_assignable_v<T&, const optional<U>&&>` is `false`.

25 *Effects*: See . The result of the expression `rhs.has_value()` remains unchanged.

Table 4: `optional::operator=(optional<U>&&)` effects [tab:optional.assign.move.templ]

	`*this` **contains a value**	`*this` **does not contain a value**
`rhs` **contains a value**	assigns `std::move(*rhs)` to the contained value	direct-non-list-initializes the contained value with `std::move(*rhs)`
`rhs` **does not contain a value**	destroys the contained value by calling `val->T::~T()`	no effect

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

27 *Returns*: `*this`.

28 *Remarks*: If any exception is thrown, the result of the expression `this->has_value()` remains unchanged. If an exception is thrown during the call to `T`'s constructor, the state of `*rhs.val` is determined by the exception safety guarantee of `T`'s constructor. If an exception is thrown during the call to `T`'s assignment, the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `T`'s assignment.

```
template<class... Args> constexpr T& emplace(Args&&... args);
```

29 *Mandates*: `is_constructible_v<T, Args...>` is `true`.

30 *Effects*: Calls `*this = nullopt`. Then direct-non-list-initializes the contained value with `std::forward<Args>(args)...`.

31 *Postconditions*: `*this` contains a value.

32 *Returns*: A reference to the new contained value.

33 *Throws*: Any exception thrown by the selected constructor of `T`.

34 *Remarks*: If an exception is thrown during the call to `T`'s constructor, `*this` does not contain a value, and the previous `*val` (if any) has been destroyed.

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

35 *Constraints*: `is_constructible_v<T, initializer_list<U>&, Args...>` is `true`.

36 *Effects*: Calls `*this = nullopt`. Then direct-non-list-initializes the contained value with `il, std::forward<Args>(args)...`.

37 *Postconditions*: `*this` contains a value.

38 *Returns*: A reference to the new contained value.

39 *Throws*: Any exception thrown by the selected constructor of `T`.

40 *Remarks*: If an exception is thrown during the call to `T`'s constructor, `*this` does not contain a value, and the previous `*val` (if any) has been destroyed.

�.�.3.5 Swap [optional.swap]

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

1 *Mandates*: `is_move_constructible_v<T>` is `true`.

2 *Preconditions*: `T` meets the *Cpp17Swappable* requirements.

3 *Effects*: See .

4 *Throws*: Any exceptions thrown by the operations in the relevant part of .

5 *Remarks*: The exception specification is equivalent to:

```
is_nothrow_move_constructible_v<T> && is_nothrow_swappable_v<T>
```

Table 5: `optional::swap(optional&)` effects [tab:optional.swap]

	`*this` contains a value	**`*this` does not contain a value**
`rhs` contains a value	calls `swap(*(*this), *rhs)`	direct-non-list-initializes the contained value of `*this` with `std::move(*rhs)`, followed by `rhs.val->T::~T()`; postcondition is that `*this` contains a value and `rhs` does not contain a value
`rhs` does not contain a value	direct-non-list-initializes the contained value of `rhs` with `std::move(*(*this))`, followed by `val->T::~T()`; postcondition is that `*this` does not contain a value and `rhs` contains a value	no effect

6 If any exception is thrown, the results of the expressions `this->has_value()` and `rhs.has_value()` remain unchanged. If an exception is thrown during the call to function `swap`, the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `swap` for lvalues of `T`. If an exception is thrown during the call to `T`'s move constructor, the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `T`'s move constructor.

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

```
using iterator       = implementation-defined;
using const_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` and `const T&` for `const_iterator`.

2 All requirements on container iterators apply to `optional::iterator` and `optional::const_iterator` as well.

3 Any operation that initializes or destroys the contained value of an optional object invalidates all iterators into that object.

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

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

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

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

�.�.3.7 Observers [optional.observe]

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

1 *Preconditions*: `*this` contains a value.

2 *Returns*: `val`.

3 *Remarks*: These functions are constexpr functions.

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

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

4 *Preconditions*: `*this` contains a value.

5 *Returns*: `*val`.

6 *Remarks*: These functions are constexpr functions.

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

7 *Preconditions*: `*this` contains a value.

8 *Effects*: Equivalent to: `return std::move(*val);`

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

9 *Returns*: `true` if and only if `*this` contains a value.

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

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

11 *Returns*: `true` if and only if `*this` contains a value.

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

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

13 *Effects*: Equivalent to:

```
return has_value() ? *val : throw bad_optional_access();
```

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

14 *Effects*: Equivalent to:

```
return has_value() ? std::move(*val) : throw bad_optional_access();
```

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

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

16 *Effects*: Equivalent to:

```
return has_value() ? **this : static_cast<T>(std::forward<U>(v));
```

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

17 *Mandates*: `is_move_constructible_v<T> && is_convertible_v<U&&, T>` is `true`.

18 *Effects*: Equivalent to:

```
return has_value() ? std::move(**this) : static_cast<T>(std::forward<U>(v));
```

�.�.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 reference or non-array object type other than in_place_t or nullopt_t. 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 reference or non-array object type other than in_place_t or nullopt_t. 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]

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

1 *Effects*: If `*this` contains a value, calls `val->T::~T()` to destroy the contained value; otherwise no effect.

2 *Postconditions*: `*this` does not contain a value.

�.�.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
      using const_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;
      constexpr optional(optional&& rhs) noexcept      = default;
      template <class U = T> constexpr explicit(see below) optional(U&& u) noexcept;
      template <class U> constexpr explicit(see below) optional(const optional<U>& rhs) noexcept;

      constexpr ~optional() = default;

      // �.�.4.3, 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>
        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) = delete;
      template <class U> constexpr optional& emplace(U&& u) noexcept;

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

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

}
```

�.�.4.2 Constructors [optionalref.ctor]

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

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

2 *Remarks*: For all T& these constructors are constexpr constructors.

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

3 *Constraints*:

(3.1) — !*is-derived-from-optional*<decay_t<U>> is true

4 *Mandates*:

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

(4.2) — is_lvalue_reference<U>::value> is true

5 *Effects*: Direct-non-list-initializes val with addressof(u).

6 *Postconditions*: *this refers to a value.

7 *Remarks*: The expression inside explicit is equivalent to:

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

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

8 *Mandates*:

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

9 *Postconditions*: If rhs.has_value() is true, *this refers to the same value, otherwise, *this does not refer to a value. *Remarks*: The expression inside explicit is equivalent to:

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

�.�.4.3 Assignment [optionalref.assign]

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

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

2 *Returns*: *this.

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

3 *Constraints*:

(3.1) — !*is-derived-from-optional*<decay_t<U>> is true

4 *Mandates*:

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

(4.2) — is_lvalue_reference<U>::value> is true

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

6 *Postconditions*: *this refers to a value.

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

7 *Mandates*:

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

(7.2) — is_lvalue_reference<U>::value> is true

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

9 *Returns*: *this.

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

10 *Constraints*:

(10.1) — !*is-derived-from-optional*<decay_t<U>> is true

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

12 *Postconditions*: *this refers to a value.

�.�.4.4 Swap [optionalref.swap]

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

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

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

```
using iterator = implementation-defined;
using const_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 and const T& for const_iterator.

2 All requirements on container iterators apply to optional::iterator and optional::const_iterator as well.

```
constexpr iterator begin() noexcept;
constexpr const_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() noexcept;
constexpr const_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*: *this refers to a value.

3 *Returns*: *val.

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

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

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

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

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

6 *Effects*: Equivalent to:

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

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

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

8 *Effects*: Equivalent to: `return has_value() ? U(**this) : U(std::forward<Args>(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:

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

```
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 *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)();
```

�.�.4.8 Modifiers [optionalref.mod]

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

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

�.�.5 Class template `optional` for rvalue reference types [optional.optionalrref]

�.�.5.1 General [optional.optionalrref.general]

```
namespace std {
  template <class T>
  class optional<T&&> {
    public:
      using value_type     = T&&;
      using iterator       = implementation-defined; // see �.�.5.5
      using const_iterator = implementation-defined; // see �.�.5.5

    public:
      // �.�.5.2, constructors
      constexpr optional() noexcept;
```

```
    constexpr optional(nullopt_t) noexcept;
    constexpr optional(const optional& rhs) noexcept = default;
    constexpr optional(optional&& rhs) noexcept      = default;
    template <class U = T> constexpr explicit(see below) optional(U&& u) noexcept;
    template <class U> constexpr explicit(see below) optional(const optional<U>& rhs) noexcept;

    constexpr ~optional() = default;

    // �.�.5.3, 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> 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) = delete;
    template <class U> constexpr optional& emplace(U&& u) noexcept;

    // �.�.5.4, swap
    constexpr void swap(optional& rhs) noexcept;

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

    // �.�.5.6, 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 T value_or(U&& u) const;

    // �.�.5.7, 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;

    // �.�.5.8, modifiers
    constexpr void reset() noexcept;

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

�.�.5.2 Constructors [optionalrref.ctor]

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

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

2 *Remarks*: For all `T&&` these constructors are constexpr constructors.

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

3 *Constraints*:

(3.1) — `!is-derived-from-optional<decay_t<U>>` is true

4 *Mandates*:

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

5 *Effects*: Direct-non-list-initializes val with addressof(u).

6 *Postconditions*: *this refers to a value.

7 *Remarks*: The expression inside explicit is equivalent to:

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

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

8 *Mandates*:

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

9 *Postconditions*: If rhs.has_value() is true, *this refers to the same value, otherwise, *this does not refer to a value. *Remarks*: The expression inside explicit is equivalent to:

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

�.�.5.3 Assignment [optionalrref.assign]

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

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

2 *Returns*: *this.

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

3 *Constraints*:

(3.1) — !*is-derived-from-optional*<decay_t<U>> is true

4 *Mandates*:

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

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

6 *Postconditions*: *this refers to a value.

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

7 *Mandates*:

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

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

9 *Returns*: *this.

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

10 *Constraints*:

(10.1) — !*is-derived-from-optional*<decay_t<U>> is true

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

12 *Postconditions*: *this refers to a value.

�.�.5.4 Swap [optionalrref.swap]

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

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

�.�.5.5 Iterator support [optionalrref.iterators]

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

```
using const_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` and `const T&` for `const_iterator`.

2 All requirements on container iterators apply to `optional::iterator` and `optional::const_iterator` as well.

```
constexpr iterator begin() noexcept;
constexpr const_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() noexcept;
constexpr const_iterator end() const noexcept;
```

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

�.�.5.6 Observers [optionalrref.observe]

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

1 *Returns*: `val`.

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

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

3 *Returns*: `std::move(*val)`.

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

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

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

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

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

6 *Effects*: Equivalent to:

```
if (has_value())
  return std::move(*val);
throw bad_optional_access();
```

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

7 *Mandates*:

(7.1) — `is_copy_constructible_v<T>` is `true`

(7.2) — `is_convertible_v<decltype(u), T>` is `true`

8 *Effects*: Equivalent to: `return has_value() ? std::move(*val) : std::forward<U>(u);`

�.�.5.7 Monadic operations [optionalrref.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()) : 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 *Returns*: If *this refers to a value, an optional<U> object whose refered to value is the result of invoke(forward<F>(f), std::move(*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 std::move(*val)
else
    return forward<F>(f)();
```

�.�.5.8 Modifiers [optionalrref.mod]

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

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

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

```
struct nullopt_t{see below};
inline constexpr nullopt_t nullopt(unspecified);
```

1 The struct nullopt_t is an empty class type used as a unique type to indicate the state of not containing a value for optional objects. In particular, optional<T> has a constructor with nullopt_t as a single argument; this indicates that an optional object not containing a value shall be constructed.

2 Type nullopt_t shall not have a default constructor or an initializer-list constructor, and shall not be an aggregate.

�.�.7 Class bad_optional_access [optional.bad.access]

```
namespace std {
  class bad_optional_access : public exception {
  public:
    // see ?? for the specification of the special member functions
    const char* what() const noexcept override;
  };
}
```

1 The class bad_optional_access defines the type of objects thrown as exceptions to report the situation where an attempt is made to access the value of an optional object that does not contain a value.

```
const char* what() const noexcept override;
```

2 *Returns*: An implementation-defined NTBS.

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

```
template<class T, class U> constexpr bool operator==(const optional<T>& x, const optional<U>& y);
```

1 *Constraints*: The expression *x == *y is well-formed and its result is convertible to bool.

[*Note 1*: T need not be *Cpp17EqualityComparable*. — *end note*]

2 *Returns*: If x.has_value() != y.has_value(), false; otherwise if x.has_value() == false, true; otherwise *x == *y.

3 *Remarks*: Specializations of this function template for which *x == *y is a core constant expression are constexpr functions.

```
template<class T, class U> constexpr bool operator!=(const optional<T>& x, const optional<U>& y);
```

4 *Constraints*: The expression `*x != *y` is well-formed and its result is convertible to `bool`.

5 *Returns*: If `x.has_value() != y.has_value()`, `true`; otherwise, if `x.has_value() == false`, `false`; otherwise `*x != *y`.

6 *Remarks*: Specializations of this function template for which `*x != *y` is a core constant expression are constexpr functions.

```
template<class T, class U> constexpr bool operator<(const optional<T>& x, const optional<U>& y);
```

7 *Constraints*: `*x < *y` is well-formed and its result is convertible to `bool`.

8 *Returns*: If `!y`, `false`; otherwise, if `!x`, `true`; otherwise `*x < *y`.

9 *Remarks*: Specializations of this function template for which `*x < *y` is a core constant expression are constexpr functions.

```
template<class T, class U> constexpr bool operator>(const optional<T>& x, const optional<U>& y);
```

10 *Constraints*: The expression `*x > *y` is well-formed and its result is convertible to `bool`.

11 *Returns*: If `!x`, `false`; otherwise, if `!y`, `true`; otherwise `*x > *y`.

12 *Remarks*: Specializations of this function template for which `*x > *y` is a core constant expression are constexpr functions.

```
template<class T, class U> constexpr bool operator<=(const optional<T>& x, const optional<U>& y);
```

13 *Constraints*: The expression `*x <= *y` is well-formed and its result is convertible to `bool`.

14 *Returns*: If `!x`, `true`; otherwise, if `!y`, `false`; otherwise `*x <= *y`.

15 *Remarks*: Specializations of this function template for which `*x <= *y` is a core constant expression are constexpr functions.

```
template<class T, class U> constexpr bool operator>=(const optional<T>& x, const optional<U>& y);
```

16 *Constraints*: The expression `*x >= *y` is well-formed and its result is convertible to `bool`.

17 *Returns*: If `!y`, `true`; otherwise, if `!x`, `false`; otherwise `*x >= *y`.

18 *Remarks*: Specializations of this function template for which `*x >= *y` is a core constant expression are constexpr functions.

```
template<class T, three_way_comparable_with<T> U>
  constexpr compare_three_way_result_t<T, U>
    operator<=>(const optional<T>& x, const optional<U>& y);
```

19 *Returns*: If `x && y`, `*x <=> *y`; otherwise `x.has_value() <=> y.has_value()`.

20 *Remarks*: Specializations of this function template for which `*x <=> *y` is a core constant expression are constexpr functions.

�.�.9 Comparison with `nullopt` [optional.nullops]

```
template<class T> constexpr bool operator==(const optional<T>& x, nullopt_t) noexcept;
```

1 *Returns*: `!x`.

```
template<class T> constexpr strong_ordering operator<=>(const optional<T>& x, nullopt_t) noexcept;
```

2 *Returns*: `x.has_value() <=> false`.

�.�.10 Comparison with `T` [optional.comp.with.t]

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

1 *Constraints*: The expression `*x == v` is well-formed and its result is convertible to `bool`.

[*Note 1*: `T` need not be *Cpp17EqualityComparable*. — *end note*]

2 *Effects*: Equivalent to: `return x.has_value() ? *x == v : false;`

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

3 *Constraints*: The expression `v == *x` is well-formed and its result is convertible to `bool`.

4 *Effects*: Equivalent to: `return x.has_value() ? v == *x : false;`

```
template<class T, class U> constexpr bool operator!=(const optional<T>& x, const U& v);
```

5 *Constraints*: The expression `*x != v` is well-formed and its result is convertible to `bool`.

6 *Effects*: Equivalent to: `return x.has_value() ? *x != v : true;`

```
template<class T, class U> constexpr bool operator!=(const T& v, const optional<U>& x);
```

7 *Constraints*: The expression `v != *x` is well-formed and its result is convertible to `bool`.

8 *Effects*: Equivalent to: `return x.has_value() ? v != *x : true;`

```
template<class T, class U> constexpr bool operator<(const optional<T>& x, const U& v);
```

9 *Constraints*: The expression `*x < v` is well-formed and its result is convertible to `bool`.

10 *Effects*: Equivalent to: `return x.has_value() ? *x < v : true;`

```
template<class T, class U> constexpr bool operator<(const T& v, const optional<U>& x);
```

11 *Constraints*: The expression `v < *x` is well-formed and its result is convertible to `bool`.

12 *Effects*: Equivalent to: `return x.has_value() ? v < *x : false;`

```
template<class T, class U> constexpr bool operator>(const optional<T>& x, const U& v);
```

13 *Constraints*: The expression `*x > v` is well-formed and its result is convertible to `bool`.

14 *Effects*: Equivalent to: `return x.has_value() ? *x > v : false;`

```
template<class T, class U> constexpr bool operator>(const T& v, const optional<U>& x);
```

15 *Constraints*: The expression `v > *x` is well-formed and its result is convertible to `bool`.

16 *Effects*: Equivalent to: `return x.has_value() ? v > *x : true;`

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

17 *Constraints*: The expression `*x <= v` is well-formed and its result is convertible to `bool`.

18 *Effects*: Equivalent to: `return x.has_value() ? *x <= v : true;`

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

19 *Constraints*: The expression `v <= *x` is well-formed and its result is convertible to `bool`.

20 *Effects*: Equivalent to: `return x.has_value() ? v <= *x : false;`

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

21 *Constraints*: The expression `*x >= v` is well-formed and its result is convertible to `bool`.

22 *Effects*: Equivalent to: `return x.has_value() ? *x >= v : false;`

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

23 *Constraints*: The expression `v >= *x` is well-formed and its result is convertible to `bool`.

24 *Effects*: Equivalent to: `return x.has_value() ? v >= *x : true;`

```
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>& x, const U& v);
```

25 *Effects*: Equivalent to: `return x.has_value() ? *x <=> v : strong_ordering::less;`

�.�.11 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`.

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

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

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

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

4 *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);
```

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

�.�.12 Hash support [optional.hash]

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

1 The specialization `hash<optional<T>>` is enabled if and only if `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`.

�.�.13 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>
```

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/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.

[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]

  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 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) = delete;

    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
};

//  \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 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>
    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;
}

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

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

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

    // \ref{optionalrref.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) = delete;

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

    // \ref{optionalrref.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{optionalrref.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{optionalrref.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
};

//  \rSec3[optionalrref.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 U>
    requires(!detail::is_optional<decay_t<U>>)
constexpr optional<T&&>::optional(U&& u) noexcept : value_(addressof(u)) {
    static_assert(is_constructible_v<add_rvalue_reference_t<T>, U>,
                  "Must be able to bind U to T&&");
}
```

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

// \rSec3[optionalrref.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_rvalue_reference_t<T>, U>,
                  "Must be able to bind U to T&&");
    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_rvalue_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>
    requires(!detail::is_optional<decay_t<U>>)
constexpr optional<T&&>& optional<T&&>::emplace(U&& u) noexcept {
    return *this = std::forward<U>(u);
}

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

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

// \rSec3[optionalrref.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[optionalrref.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 std::move(*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 std::move(*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 bind u to T");
    return has_value() ? std::move(*value_) : std::forward<U>(u);
}

//   \rSec3[optionalrref.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), std::move(*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), std::move(*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() ? std::move(*value_) : std::forward<F>(f)();
}

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