

Expected Over References

Steve Downey <sdowney@gmail.com>

Document #: D4280R0
Date: 2026-09-02
Project: Programming Language C++
Audience: LEWG

Abstract

A function that can fail and otherwise returns a reference has no good return type today. `expected<T*, E>` lets a null pointer through as a success, and `expected<reference_wrapper<T>, E>` exposes the wrapper at every use. An error that is owned elsewhere, by a diagnostics engine or an error registry, has the same problem in the error position. We propose that `std::expected` hold an lvalue reference in the value position, the error position, or both.

This takes one new partial specialization, `expected<T&, E>`, for a reference value; a relaxation of the primary `expected<T, E>` and of `expected<void, E>` to admit an lvalue-reference error type, which today they reject; and a new `unexpected<E&>` to carry one. The design follows `optional<T&>` [P2988R12]. A reference is held as a pointer. Assignment rebinds; it never assigns through to the referent. `const` is shallow: the binding is fixed, the referent is not. Constructors that would bind a reference to a temporary are deleted, with a diagnostic.

Contents

1	Before / After	1
2	The proposal	3
3	Motivation	5
4	Prior art and precedent	5
5	Design choices and decisions	6
6	Anticipated objections	8
7	Impact on the standard	8
8	Wording	9
9	Acknowledgments	39
	References	39

Changes Since Last Version

— R0 Initial revision.

1 Before / After

The value case first. A lookup that returns a reference to an element of a container, and can fail, has no satisfying spelling today.

1.1 A fallible lookup that returns a reference

The conventional idiom is a raw pointer, or an out-parameter. Neither carries the error reason.

<pre>Cat* find_cat(std::string_view name); Cat* cat = find_cat("Fido"); if (cat != nullptr) { return doit(*cat); } // error reason lost</pre>	<pre>auto find_cat(std::string_view name) -> std::expected<Cat&, Error>; auto cat = find_cat("Fido"); return cat.and_then(doit);</pre>
--	--

1.2 expected<Cat*, E> as a substitute

Returning `expected<Cat*, E>` is the common workaround. It forces an extra dereference on the happy path, and it still lets a null pointer escape as a “success”.

<pre>auto find_cat(std::string_view name) -> std::expected<Cat*, Error>; auto cat = find_cat("Fido"); if (cat && *cat) { // two checks return doit(**cat); // two derefs }</pre>	<pre>auto find_cat(std::string_view name) -> std::expected<Cat&, Error>; auto cat = find_cat("Fido"); return cat.and_then(doit);</pre>
--	--

1.3 expected<reference_wrapper<Cat>, E>

`std::reference_wrapper` closes the null hole but is verbose, exposes the wrapper rather than the reference as the value type, wants `.get()` where the reference is used, and requires adapters when composing monadic operations.

<pre>auto find_cat(std::string_view name) -> std::expected< std::reference_wrapper<Cat>, Error>; auto cat = find_cat("Fido"); if (cat) { return doit(cat->get()); }</pre>	<pre>auto find_cat(std::string_view name) -> std::expected<Cat&, Error>; auto cat = find_cat("Fido"); return cat.and_then(doit);</pre>
--	--

1.4 Monadic composition is uniform

The monadic API works the same whether `T` is a value or a reference, so a chain of fallible steps never has to leave `expected`.

<pre>Cat* cat = find_cat("Fido"); if (!cat) return make_error(); std::string* name = get_name(*cat); if (!name) return make_error(); return process(*name);</pre>	<pre>return find_cat("Fido") .and_then(get_name) .and_then(process);</pre>
---	--

1.5 An error you do not own

The error channel has the same problem, and it is not hypothetical. A parser reports through a long-lived diagnostics engine; a driver reports through an error registry. The error is an existing object elsewhere, not a fresh value to copy into the `expected`. Today the choices are to copy it or to smuggle a pointer through `E`. `expected<T, E&>`, carrying its error by `unexpected<E&>`, says what is meant.

<pre>// copies the whole diagnostic, or // hides a pointer inside the error type std::expected<Node*, Diagnostic> parse(Source&, DiagnosticEngine&);</pre>	<pre>std::expected<Node&, Diagnostic&> parse(Source& s, DiagnosticEngine& d) { if (bad(s)) return std::unexpected(d.report(s)); return node_for(s); }</pre>
--	---

Nothing in the “after” column is new API. Each entry is the existing `expected` API with a reference where an object used to be.

2 The proposal

`std::expected<T, E>` [P0323R12], added in C++ 2023, holds either a value of type `T` or an error of type `E`. Like `std::optional` before it, the primary template forbids reference types; the working draft instead requires that a program instantiating `expected<T, E>` use `void` or a complete non-array object type for `T`; otherwise the program is ill-formed. Reference values need a specialization; reference errors do not. In full, the proposal is:

- a new partial specialization `expected<T&, E>` for a reference value, in which `E` may be an object type or an lvalue reference;
- a relaxation of the primary `expected<T, E>` and of `expected<void, E>`, whose error type `E` may now be an lvalue reference where today it may not; and
- a new partial specialization `unexpected<E&>`, so that `unexpected` can carry a reference error.

An error you do not own is stored as the pointer already inside `unexpected<E&>`, so permitting a reference `E` means relaxing a constraint. The primary `expected<T, E>` and `expected<void, E>` are otherwise unchanged. A reference is stored as a pointer (`T&` as a `T*`, a reference error as the pointer within `unexpected<E&>`), and there is no new state and no new vocabulary; the observers, the modifiers, the monadic operations, and the comparisons keep their names and their meanings.

An illustrative synopsis of the one new specialization follows. It is the primary template with the value slot turned into a pointer and the operations a reference cannot support deleted, and nothing else.

```
namespace std {
template<class T, class E>
class expected<T&, E> {
public:
    using value_type      = T&;
    using error_type      = E;
    using unexpected_type = unexpected<E>;
    template<class U>
    using rebind = expected<U, error_type>;

    // constructors
    constexpr expected() = delete;
    constexpr expected(const expected&);
    constexpr expected(expected&&) noexcept(see below);
    template<class U = T>
    constexpr explicit(see below) expected(U&&);
    template<class U>
    constexpr expected(U&&) = delete;
    template<class U, class G>
    constexpr explicit(see below) expected(const expected<U&, G>&);
    template<class U, class G>
    constexpr explicit(see below) expected(expected<U&, G>&&);
    template<class G>
    constexpr explicit(see below) expected(const unexpected<G>&);
    template<class G>
    constexpr explicit(see below) expected(unexpected<G>&&);
    template<class... Args>
    constexpr expected(in_place_t, Args&&...) = delete;
    template<class... Args>
    constexpr explicit expected(unexpect_t, Args&&...);
    template<class U, class... Args>
    constexpr explicit expected(unexpect_t, initializer_list<U>, Args&&...);

    constexpr ~expected();

    // assignment
    constexpr expected& operator=(const expected&);
    constexpr expected& operator=(expected&&) noexcept(see below);
    template<class U = T>
    constexpr expected& operator=(U&&);
```

```

template<class G>
constexpr expected& operator=(const unexpected<G>&);
template<class G>
constexpr expected& operator=(unexpected<G>&&);
template<class U = T>
constexpr T& emplace(U&&) noexcept(see below);

constexpr void          swap(expected&) noexcept(see below);
friend constexpr void swap(expected&, expected&) noexcept(see below);

// 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&;
constexpr T&          value() &&;
constexpr const E&    error() const& noexcept;
constexpr E&          error() & noexcept;
constexpr const E&&   error() const&& noexcept;
constexpr E&&         error() && noexcept;
template<class U = remove_cv_t<T>>
constexpr remove_cv_t<T> value_or(U&&) const;
template<class G = remove_cv_t<remove_reference_t<E>>>
constexpr remove_cv_t<remove_reference_t<E>> error_or(G&&) const&;
template<class G = remove_cv_t<remove_reference_t<E>>>
constexpr remove_cv_t<remove_reference_t<E>> error_or(G&&) &&;

// monadic operations
template<class F>
constexpr auto and_then(F&&) &;
template<class F>
constexpr auto and_then(F&&) &&;
template<class F>
constexpr auto and_then(F&&) const&;
template<class F>
constexpr auto and_then(F&&) const&&;
template<class F>
constexpr auto or_else(F&&) &;
template<class F>
constexpr auto or_else(F&&) &&;
template<class F>
constexpr auto or_else(F&&) const&;
template<class F>
constexpr auto or_else(F&&) const&&;
template<class F>
constexpr auto transform(F&&) &;
template<class F>
constexpr auto transform(F&&) &&;
template<class F>
constexpr auto transform(F&&) const&;
template<class F>
constexpr auto transform(F&&) const&&;
template<class F>
constexpr auto transform_error(F&&) &;
template<class F>
constexpr auto transform_error(F&&) &&;
template<class F>
constexpr auto transform_error(F&&) const&;
template<class F>
constexpr auto transform_error(F&&) const&&;

```

```

// equality operators
template<class T2, class E2>
friend constexpr bool operator==(const expected&, const expected<T2, E2>&);
template<class T2>
friend constexpr bool operator==(const expected&, const T2&);
template<class E2>
friend constexpr bool operator==(const expected&, const unexpected<E2>&);

private:
    bool has_val;    // exposition only
    union {
        T* val;      // exposition only
        unexpected<E> unex; // exposition only
    };
};
}

```

The Beman reference implementation is available online [[Downey_beman_expected](#)].

3 Motivation

Functions that return a reference to an element of a container are common in performance-sensitive code, because they do not copy. When such a function can fail, there is no good return type today. The Before/After tables above show the three current spellings and what each loses.

`expected<T&, E>` fills the gap. It participates in `and_then`, `transform`, `or_else`, and `transform_error` with no adapter.

`std::optional<T&>` was accepted as [[P2988R12](#)]. Its motivation observed:

Thus, we expect future papers to propose `std::expected<T&, E>` and `std::variant` with the ability to hold references.

This is the first of those two. It is also more than the sentence asked for: `E` is a template parameter exactly as `T` is, and an error you do not own is as ordinary as a value you do not own. Supporting references in the value position but not the error position would be an arbitrary asymmetry. So the proposal covers both: the value position with a specialization, the error position by letting `E` be a reference in every one of them.

There is a principled reason the standard disallowed references in its sum types: assignment means something different for a reference than for a value. `optional<T>` has value semantics and assignment copies; `optional<T&>` rebinds. The observable behavior differs. However, the same difference already holds between `int` and `int*`, and between `tuple<int>` and `tuple<int&>`. The standard already stores references in `std::tuple`, `std::pair`, and `std::reference_wrapper`. A fallible sum type is not a special case that warrants an exception.

`optional<T>` is `variant<T, monostate>`; `expected<T, E>` is `variant<T, E>`. `tuple` and `pair` can hold references; `variant` cannot. The missing sum-type support is a separate and larger defect, and it should not block a feature that can be specified precisely now by following `optional<T&>`.

4 Prior art and precedent

The design is not invented here. `optional<T&>` [[P2988R12](#)] settled the hard questions (rebind versus assign-through, shallow versus deep `const`, dangling prevention) after years of debate, and this paper copies its answers rather than relitigating them. Where the two designs diverge, it is only because `expected` carries an error where `optional` carries nothing.

The choice of rebind semantics rests on JeanHeyd Meneide's survey of reference-holding optionals [[P1683R0](#), [REFBIND](#)]: every production implementation that shipped assign-through was later withdrawn or rewritten. The result is not a matter of taste. All other semantics are worse.

The wider precedent is that the standard already holds references in its product types. `tuple<int&>`, `pair<int&, int>`, and `reference_wrapper<T>` exist and are used. What is missing is references in the sum types, and this paper closes half of that gap; `variant<T&>` is the other half, tracked separately.

5 Design choices and decisions

The decisions that shape the interface are argued here.

5.1 Rebind on assignment, never assign-through

Assignment to an engaged `expected<T&, E>` rebinds the internal pointer. It does not assign through to the referent.

```
int a = 1, b = 2;
std::expected<int&, int> e(a);
e = std::expected<int&, int>(b); // e now refers to b; a is still 1
```

This matches `std::reference_wrapper::operator=`, which rebinds the wrapper rather than assigning through to its referent. [P2988R12] settled the same question for `optional<T&>`, and the survey behind it [P1683R0] records assign-through implementations that were abandoned. `emplace` rebinds too, and `swap` exchanges the two pointers and leaves the referents alone.

5.2 Shallow const

A `const expected<T&, E>` yields a non-const `T&` from `operator*`. The binding is `const`; the referent is not. The model is `T* const`, not `const T*`, and it matches `reference_wrapper` and `optional<T&>`. A caller who wants deep `const` spells `expected<const T&, E>`.

A `const&` parameter of this type no longer promises “I will not touch your data.” A reference never made that promise, and neither does a `T* const`; the mistake is in reading `const` on a handle as `const` on the referent.

5.3 Dangling constructors are deleted, with a diagnostic

Every constructor that would bind `T&` (or `E&`) to a temporary is deleted, guarded by `reference_constructs_from_temporary_v` [P2255R2]. This follows the `tuple` and `pair` precedent: make the dangling call ill-formed, rather than silently dropping the candidate and letting a worse overload win.

One wrinkle produces two diagnostics for what looks like one error. A non-`const int&` simply cannot bind a temporary at all, so `expected<int&, E>(42)` fails as “no matching function.” A `const int&` *can* bind a prvalue, so `expected<const int&, E>(42)` is where the deleted overload fires and the message is seen. Both are correct rejections. They do not read identically, and a reviewer should expect both forms.

The detection is deliberately type-based. For a forwarding constructor, a prvalue and an xvalue deduce the same source type; the source expression’s provenance is no longer available to the constraint. Accepting xvalues while rejecting prvalues would therefore require information this interface cannot recover, so both are conservatively rejected.

5.4 References in the error position, by relaxation not specialization

The error type is a template parameter, and errors are routinely owned elsewhere: a diagnostics engine, an error registry, a long-lived context object. So a reference error must be expressible. It does not, however, need a class of its own: an `expected` already stores its error as an `unexpected<E>`, and `unexpected<E&>` holds a pointer, so admitting a reference `E` only requires relaxing a constraint. Every `expected` template (the primary, the `void` specialization, and the new `expected<T&, E>`) changes its guard from “`E` is not a reference” to “`E` is not an *rvalue* reference,” and reference errors work throughout. Not being able to express `expected<T, E&>` while `expected<T&, E>` works would be inconsistent, hostile, and strictly worse than disallowing both.

5.5 unexpected<E&>, and the reference-only rule

To construct an `expected` with a reference error, `unexpected` has to be able to hold one, so `unexpected<E&>` is part of the proposal. It stores a pointer to an external object.

When the error type is a reference, construction from `unexpected<G>` is offered *only when G is itself a reference*, and deleted when `G` is a value:

```
int err = 42;
std::expected<int, int&> a(std::unexpected, err); // OK: binds to err
std::expected<int, int&> b(std::unexpected<int&>(err)); // OK: source holds a reference
std::expected<int, int&> c(std::unexpected(42)); // deleted: value would dangle
```

The reason is ownership. An `unexpected<E&>` refers to an object it does not own, so binding `E&` to its `error()` cannot dangle, even when the `unexpected` is a temporary. An `unexpected<G>` for a value `G` owns that `G`, and binding `E&` to it dangles the instant that, usually temporary, `unexpected` is destroyed. So the safe case is allowed and the dangling case is deleted.

Deleting *all* `unexpected<G>` construction for a reference `E` would throw out the safe `unexpected<E&>` case with the dangerous one. Rebinding assignment from `unexpected<E&>` follows the same reference-only rule, and repoints the stored pointer rather than assigning through to the old referent. The value-`G` cliff (`unexpected<int&>` works, `unexpected(42)` does not) is a likely question, and the deleted overload's message names it.

5.6 `value_or` returns by value; `error_or` mirrors it

`value_or` returns `T` by value, as `optional<T&>::value_or` does [P2988R12]: a reference return would force the alternative to be a `T&` or something convertible to one, which is too restrictive to be useful. `error_or` returns the error by value on the same reasoning. Callers who genuinely want a reference-returning fallback have the free-function forms, `reference_or`, from the unpublished draft [D4270R0], which work over all the nullable types at once.

For `expected<void, E&>` there is no value. `value_or` is `= deleted` with a message, so the error names the reason instead of leaving the reader to infer it from “no member named `value_or`.”

5.7 No default constructor, no in-place value, `emplace` rebinds

An engaged `expected<T&, E>` must hold a real reference; there is no null reference, so there is no default constructor. `expected(in_place, ...)` is deleted; there is nowhere to construct a `T`. `emplace` rebinds the pointer, which is the only thing it can do; nothing is being “emplaced,” but the name is kept for parallelism with the primary template.

5.8 Monadic operations are uniform

`and_then`, `transform`, `or_else`, and `transform_error` are provided on every specialization. On a reference specialization the callable is always handed `T&` (or the error `E&`), regardless of the value category of the `expected` it is called on: the stored thing is a pointer, and an rvalue `expected` does not make the borrowed object an rvalue. The four ref-qualified overloads are kept for parity with the primary template even though, for a reference, they do the same work.

5.9 Shallow conversions must not steal

Converting a reference-holding `expected` to a value-holding one copies the referent; it never moves out of it, even from an rvalue source. `std::move` on an `expected<T&, E>` moves the handle, a pointer, not the object it points at. This is the `boost::optional<T&>` pitfall, stated as a guarantee callers may rely on:

```
std::string s = "bar";
std::expected<std::string&, int> r{s};
std::expected<std::string, int> v{std::move(r)};    // copies; s == "bar" afterwards
```

The internal rule that secures this is move-the-wrapper-then-access. For example, use `*std::move(r)` or `std::move(r).error()`. Never use access-then-move: `std::move(*r)` would force an rvalue onto a reference the wrapper does not own, and steal from the caller's object.

5.10 Deleted functions carry messages

Every deleted operation says what is wrong and what to do instead. For example, the deleted default constructor explains that `T&` cannot be null. Deleted-function messages [P2573R2] are C++ 2026, so every implementation of this proposal has them. The wording does not specify the message text.

5.11 The held `unexpected<E>` is exposition-only; the observable behavior is keyed on `E`

Storing the error as `unexpected<E>` is a specification device, not a representation mandate. It supplies the reference case (`unexpected<E&>` holds a pointer) without a new specialization, and that is its only purpose. In particular it must not reach an existing implementation and tell it to change what `expected<T, E>` stores.

So the exposition-only member is `unexpected<E>`, but every observable property is stated in terms of `E`: the *Constraints*, the *Mandates*, the exception specifications, and the triviality of each special member function read exactly as they do in the current `T-and-E` wording. An implementation that stores `E` directly, as every

shipping one does, is conforming unchanged. Its layout, its name mangling, and its trait answers do not move. The design targets ABI stability for existing implementations.

This is safe because `unexpected<E>` is a transparent wrapper over `E`: for an object `E` its copy, move, assignment, destruction, exception specifications, and triviality all track `E`'s own. The one place the language lets such a wrapper diverge — a defaulted move that is defined as deleted is ignored by overload resolution, where a user-deleted move is not — is reconciled by `expected`'s own fall-back from a constrained-out move to its copy, and lands on the same answer. The equivalence was checked trait by trait against `std::expected` across a battery of awkward error types.

If an observable difference between the `unexpected<E>` rendering and a rendering with separate `T` and `E` members is ever found, that is a defect in this proposal, and its resolution is already decided: the behavior is whatever the `T-and-E` rendering gives.

5.12 `expected<T, E>` is trivially copyable when `T` and `E` are

A drive-by. `expected<int, int>` is not trivially copyable today, because `expected`'s copy and move assignment are specified as non-trivial operations even when every component is trivial. There is no semantic reason for this. The assignment of a sum of trivially copyable, trivially assignable, trivially destructible types can be defaulted, and then the whole `expected` is trivially copyable: it can be passed in registers and relocated with `memcpy`.

“Non-trivial by specification” is not a reason; it is the absence of one. This proposal specifies the special member functions of `expected` to be trivial when `T` and `E` make them trivial, matching the intuition that a sum of trivial types is trivial. The change is a widening: types that were trivially copyable stay so, and `expected<int, int>` joins them.

6 Anticipated objections

6.1 “Use `reference_wrapper` instead”

It works, and it is what people write today. But `.get()` litters every use, the wrapper remains visible in the type, and monadic composition needs adaptation before a callable receives `T&`. Its assignment correctly rebinds; the other interface friction is what the reference specialization retires.

6.2 “References in sum types were banned for a reason”

They were, and the reason is real: assignment differs between a value and a reference. It also differs for `int` versus `int*`, and for `tuple<int>` versus `tuple<int&>`, both of which the standard ships. The difference is a property of the type the programmer asked for, not a defect in the container. `optional<T&>` decided this; `expected` should decide it the same way.

6.3 “`expected<T*, E>` is good enough”

It is two checks and two dereferences on the happy path, and it lets a null `T*` escape through the success channel as though it were a value. The type says less than the code means.

6.4 “Wait for `variant<T&>` and do it all at once”

`tuple` and `pair` can hold references today; `variant` cannot. Extending `variant` is a separate, larger job. `expected<T&, E>` can be specified right now by following `optional<T&>` line for line. A clean, ready feature should not wait on an unrelated one.

7 Impact on the standard

A pure library extension. No core-language change. The only well-formed programs affected are those that asserted a reference error, or the `expected<T&, E>` specialization, is rejected; those were excluding the feature on purpose. The change adds the `expected<T&, E>` and `unexpected<E&>` specializations, and widens the error type of the existing `expected` templates from an object type to an object type or an lvalue reference.

8 Wording

The proposed wording is a marked edit of the `[expected]` subclause of the current working draft, reproduced below, and of two feature-test macros in `[version.syn]`: additions are shown [like this](#) and deletions ~~like this~~. The baseline is checked in verbatim and edited only in the copy, so the marks are exactly the proposed change and nothing else.

The `[expected]` edits are of two kinds. A new `[expected.ref]` subclause specifies `expected<T&, E>`, following the `optional<T&>` wording of [\[P2988R12\]](#): a synopsis with the value slot a pointer, the deleted default and in-place constructors, the rebind-semantic assignment, and the shallow-`const` observers. Marked edits in place widen the error type of the primary `expected<T, E>` and of `expected<void, E>` to admit an lvalue reference, and add `unexpected<E&>` with the reference-only construction rule above. The marked edits included here are the proposed changes.

In `[version.syn]`, header `<version>` synopsis, update the values of the following macros, with the value selected by the editor to reflect the date of adoption of this paper:

```
#define __cpp_lib_expected          202606L20XXXXL // also in <expected>
#define __cpp_lib_freestanding_expected 202311L20XXXXL // freestanding, also in <expected>
```

8.1 Expected objects

[expected]

8.1.1 General

[expected.general]

- ¹ Subclause 8.1 describes the class template `expected` that represents expected objects. An `expected<T, E>` object holds an object of type `T` or an object of type `E` and manages the lifetime of the contained objects.

8.1.2 Header `<expected>` synopsis

[expected.syn]

```
// mostly freestanding
namespace std {
    // 8.1.3, class template unexpected
    template<class E> class unexpected;

    // 8.1.4, class template bad_expected_access
    template<class E> class bad_expected_access;

    // 8.1.5, specialization for void
    template<> class bad_expected_access<void>;

    // in-place construction of unexpected values
    struct unexpect_t {
        explicit unexpect_t() = default;
    };
    inline constexpr unexpect_t unexpect{};

    // 8.1.6, class template expected
    template<class T, class E> class expected;

    // 8.1.7, partial specialization of expected for void types
    template<class T, class E> requires is_void_v<T> class expected<T, E>;
} // partially freestanding
```

8.1.3 Class template `unexpected`

[expected.unexpected]

8.1.3.1 General

[expected.un.general]

- ¹ Subclause 8.1.3 describes the class template `unexpected` that represents unexpected objects stored in `expected` objects.

```
namespace std {
    template<class E>
    class unexpected {
    public:
        // 8.1.3.2, constructors
        constexpr unexpected(const unexpected&) = default;
        constexpr unexpected(unexpected&&) = default;
        template<class Err = E>
            constexpr explicit unexpected(Err&&);
        template<class... Args>
            constexpr explicit unexpected(in_place_t, Args&&...);
        template<class U, class... Args>
            constexpr explicit unexpected(in_place_t, initializer_list<U>, Args&&...);

        constexpr unexpected& operator=(const unexpected&) = default;
        constexpr unexpected& operator=(unexpected&&) = default;

        constexpr const E& error() const & noexcept;
        constexpr E& error() & noexcept;
        constexpr const E&& error() const && noexcept;
        constexpr E&& error() && noexcept;

        constexpr void swap(unexpected& other) noexcept(see below);

        template<class E2>
            friend constexpr bool operator==(const unexpected&, const unexpected<E2>&);
    };
}
```

```

    friend constexpr void swap(unexpected& x, unexpected& y) noexcept(noexcept(x.swap(y)));

private:
    E unex;          // exposition only
};

template<class E> unexpected(E) -> unexpected<E>;
}

```

- ² A program that instantiates the definition of `unexpected` for a non-object type [other than an lvalue reference type](#), an array type, a specialization of `unexpected`, or a cv-qualified type is ill-formed.

8.1.3.2 Constructors

[expected.un.cons]

```

template<class Err = E>
constexpr explicit unexpected(Err&& e);

```

¹ *Constraints:*

- (1.1) — `is_same_v<remove_cvref_t<Err>, unexpected>` is false; and
- (1.2) — `is_same_v<remove_cvref_t<Err>, in_place_t>` is false; and
- (1.3) — `is_constructible_v<E, Err>` is true.

² *Effects:* Direct-non-list-initializes *unex* with `std::forward<Err>(e)`.

³ *Throws:* Any exception thrown by the initialization of *unex*.

```

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

```

⁴ *Constraints:* `is_constructible_v<E, Args...>` is true.

⁵ *Effects:* Direct-non-list-initializes *unex* with `std::forward<Args>(args)...`

⁶ *Throws:* Any exception thrown by the initialization of *unex*.

```

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

```

⁷ *Constraints:* `is_constructible_v<E, initializer_list<U>&, Args...>` is true.

⁸ *Effects:* Direct-non-list-initializes *unex* with `il, std::forward<Args>(args)...`

⁹ *Throws:* Any exception thrown by the initialization of *unex*.

8.1.3.3 Observers

[expected.un.obs]

```

constexpr const E& error() const & noexcept;
constexpr E& error() & noexcept;

```

¹ *Returns:* *unex*.

```

constexpr E&& error() && noexcept;
constexpr const E&& error() const && noexcept;

```

² *Returns:* `std::move(unex)`.

8.1.3.4 Swap

[expected.un.swap]

```

constexpr void swap(unexpected& other) noexcept(is_nothrow_swappable_v<E>);

```

¹ *Mandates:* `is_swappable_v<E>` is true.

² *Effects:* Equivalent to: using `std::swap`; `swap(unex, other.unex)`;

```

friend constexpr void swap(unexpected& x, unexpected& y) noexcept(noexcept(x.swap(y)));

```

³ *Constraints:* `is_swappable_v<E>` is true.

⁴ *Effects:* Equivalent to `x.swap(y)`.

8.1.3.5 Equality operator

[expected.un.eq]

```
template<class E2>
friend constexpr bool operator==(const unexpected& x, const unexpected<E2>& y);
```

1 *Mandates:* The expression `x.error() == y.error()` is well-formed and its result is convertible to `bool`.

2 *Returns:* `x.error() == y.error()`.

8.1.3.6 Partial specialization `unexpected<E&>`

[expected.un.ref]

```
template<class E>
class unexpected<E&> {
public:
    constexpr unexpected(const unexpected&) = default;
    constexpr unexpected(unexpected&&) = default;
    template<class G = E>
        constexpr explicit unexpected(G&&) noexcept;
    template<class G = E>
        constexpr explicit unexpected(in_place_t, G&&) noexcept;

    constexpr unexpected& operator=(const unexpected&) = default;
    constexpr unexpected& operator=(unexpected&&) = default;

    constexpr E& error() const noexcept;

    constexpr void swap(unexpected& other) noexcept;

    template<class E2>
        friend constexpr bool operator==(const unexpected&, const unexpected<E2>&);

    friend constexpr void swap(unexpected& x, unexpected& y) noexcept;

private:
    E* unex;           // exposition only
};
```

1 An object of type `unexpected<E&>` holds a pointer to an object of type `E`. The referenced object is not owned by the `unexpected<E&>` object.

2 A program that instantiates the definition of `unexpected<E&>` for an array type or a specialization of `unexpected` is ill-formed. Unlike the primary template, `E` may be a cv-qualified type.

```
template<class G = E>
constexpr explicit unexpected(G&& e) noexcept;
```

3 *Constraints:*

- (3.1) — `is_same_v<remove_cvref_t<G>, unexpected>` is false,
- (3.2) — `is_same_v<remove_cvref_t<G>, in_place_t>` is false,
- (3.3) — `is_constructible_v<E&, G>` is true, and
- (3.4) — `reference_constructs_from_temporary_v<E&, G>` is false.

4 *Effects:* Initializes `unex` with `addressof(static_cast<E&>(std::forward<G>(e)))`.

5 *Remarks:* A constructor for which `reference_constructs_from_temporary_v<E&, G>` is true — one that would bind `E&` to a temporary — is defined as deleted.

```
template<class G = E>
constexpr explicit unexpected(in_place_t, G&& e) noexcept;
```

6 *Constraints:* `is_constructible_v<E&, G>` is true and `reference_constructs_from_temporary_v<E&, G>` is false.

7 *Effects:* Initializes `unex` with `addressof(static_cast<E&>(std::forward<G>(e)))`.

```
constexpr E& error() const noexcept;
8     Returns: *unex.
9     Remarks: The reference returned is not const-qualified even when *this is const; the constness of
    the unexpected<E&> object does not propagate to the referenced object.

constexpr void swap(unexpected& other) noexcept;
10    Effects: Exchanges unex and other.unex.

template<class E2>
    friend constexpr bool operator==(const unexpected& x, const unexpected<E2>& y);
11    Mandates: The expression x.error() == y.error() is well-formed and its result is convertible to
    bool.
12    Returns: x.error() == y.error().

friend constexpr void swap(unexpected& x, unexpected& y) noexcept;
13    Effects: Equivalent to x.swap(y).
```

8.1.4 Class template bad_expected_access

[expected.bad]

```
namespace std {
    template<class E>
    class bad_expected_access : public bad_expected_access<void> {
    public:
        constexpr explicit bad_expected_access(E);
        constexpr const char* what() const noexcept override;
        constexpr E& error() & noexcept;
        constexpr const E& error() const & noexcept;
        constexpr E&& error() && noexcept;
        constexpr const E&& error() const && noexcept;

    private:
        E unex;           // exposition only
    };
}
```

- 1 The class template `bad_expected_access` defines the type of objects thrown as exceptions to report the situation where an attempt is made to access the value of an `expected<T, E>` object for which `has_value()` is false.

```
constexpr explicit bad_expected_access(E e);
```

- 2 *Effects:* Initializes *unex* with `std::move(e)`.

```
constexpr const E& error() const & noexcept;
constexpr E& error() & noexcept;
```

- 3 *Returns:* *unex*.

```
constexpr E&& error() && noexcept;
constexpr const E&& error() const && noexcept;
```

- 4 *Returns:* `std::move(unex)`.

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

- 5 *Returns:* An implementation-defined NTBS, which during constant evaluation is encoded with the ordinary literal encoding.

8.1.5 Class template specialization bad_expected_access<void>

[expected.bad.void]

```
namespace std {
    template<>
    class bad_expected_access<void> : public exception {
    protected:
        constexpr bad_expected_access() noexcept;
        constexpr bad_expected_access(const bad_expected_access&) noexcept;
```

```

constexpr bad_expected_access(bad_expected_access&&) noexcept;
constexpr bad_expected_access& operator=(const bad_expected_access&) noexcept;
constexpr bad_expected_access& operator=(bad_expected_access&&) noexcept;
constexpr ~bad_expected_access();

public:
    constexpr const char* what() const noexcept override;
};
}

```

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

- ¹ *Returns:* An implementation-defined NTBS, which during constant evaluation is encoded with the ordinary literal encoding.

8.1.6 Class template `expected`

[expected.expected]

8.1.6.1 General

[expected.object.general]

```

namespace std {
    template<class T, class E>
    class expected {
    public:
        using value_type = T;
        using error_type = E;
        using unexpected_type = unexpected<E>;

        template<class U>
        using rebind = expected<U, error_type>;

        // 8.1.6.2, constructors
        constexpr expected();
        constexpr expected(const expected&);
        constexpr expected(expected&&) noexcept(see below);
        template<class U, class G>
            constexpr explicit(see below) expected(const expected<U, G>&);
        template<class U, class G>
            constexpr explicit(see below) expected(expected<U, G>&&);

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

        template<class G>
            constexpr explicit(see below) expected(const unexpected<G>&);
        template<class G>
            constexpr explicit(see below) expected(unexpected<G>&&);

        template<class... Args>
            constexpr explicit expected(in_place_t, Args&&...);
        template<class U, class... Args>
            constexpr explicit expected(in_place_t, initializer_list<U>, Args&&...);
        template<class... Args>
            constexpr explicit expected(unexpect_t, Args&&...);
        template<class U, class... Args>
            constexpr explicit expected(unexpect_t, initializer_list<U>, Args&&...);

        // 8.1.6.3, destructor
        constexpr ~expected();

        // 8.1.6.4, assignment
        constexpr expected& operator=(const expected&);
        constexpr expected& operator=(expected&&) noexcept(see below);
        template<class U = remove_cv_t<T>> constexpr expected& operator=(U&&);
        template<class G>
            constexpr expected& operator=(const unexpected<G>&);
    };
}

```

```

template<class G>
    constexpr expected& operator=(unexpected<G>&&);

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

// 8.1.6.5, swap
constexpr void swap(expected&) noexcept(see below);
friend constexpr void swap(expected& x, expected& y) noexcept(noexcept(x.swap(y)));

// 8.1.6.6, observers
constexpr const T* operator->() const noexcept;
constexpr T* operator->() noexcept;
constexpr const T& operator*() const & noexcept;
constexpr T& operator*() & noexcept;
constexpr const T&& operator*() const && noexcept;
constexpr T&& operator*() && 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 const T&& value() const &&; // freestanding-deleted
constexpr T&& value() &&; // freestanding-deleted
constexpr const E& error() const & noexcept;
constexpr E& error() & noexcept;
constexpr const E&& error() const && noexcept;
constexpr E&& error() && noexcept;
template<class U = remove_cv_t<T>> constexpr T value_or(U&&) const &;
template<class U = remove_cv_t<T>> constexpr T value_or(U&&) &&;
template<class G = remove_cv_t<remove_reference_t<E>>>
    constexpr remove_cv_t<remove_reference_t<E>> error_or(G&&) const &;
template<class G = remove_cv_t<remove_reference_t<E>>>
    constexpr remove_cv_t<remove_reference_t<E>> error_or(G&&) &&;

// 8.1.6.7, 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 or_else(F&& f) &;
template<class F> constexpr auto or_else(F&& f) &&;
template<class F> constexpr auto or_else(F&& f) const &;
template<class F> constexpr auto or_else(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 auto transform_error(F&& f) &;
template<class F> constexpr auto transform_error(F&& f) &&;
template<class F> constexpr auto transform_error(F&& f) const &;
template<class F> constexpr auto transform_error(F&& f) const &&;

// 8.1.6.8, equality operators
template<class T2, class E2> requires (!is_void_v<T2>)
    friend constexpr bool operator==(const expected& x, const expected<T2, E2>& y);
template<class T2>
    friend constexpr bool operator==(const expected&, const T2&);
template<class E2>
    friend constexpr bool operator==(const expected&, const unexpected<E2>&);

private:
    bool has_val; // exposition only

```

```

    union {
        remove_cv_t<T> val;    // exposition only
        unexpected<E> unex;    // exposition only
    };
};
}

```

- ¹ Any object of type `expected<T, E>` either contains a value of type `T` or a value of type `E` nested within it. Member `has_val` indicates whether the `expected<T, E>` object contains an object of type `T`.
- ² When `has_value()` is `false`, the error is `unex.error()`. The error is held as an `unexpected<E>`, and not as an `E`, so that `E` may be an lvalue reference type: an `E&` cannot be a union member, whereas `unexpected<E&>` holds a pointer to an external object.
- ³ A type `T` is a *valid value type for `expected`*, if `remove_cv_t<T>` is `void` or a complete non-array object type that is not `in_place_t`, `unexpected_t`, or a specialization of `unexpected`. A program which instantiates class template `expected<T, E>` with an argument `T` that is not a valid value type for `expected` is ill-formed. A program that instantiates the definition of the template `expected<T, E>` with a type for the `E` parameter that is not a valid template argument for `unexpected` is ill-formed.
- ⁴ When `T` is not `cv void`, it shall meet the *Cpp17Destructible* requirements (). [If `E` is not an lvalue reference type](#), `E` shall meet the *Cpp17Destructible* requirements.

8.1.6.2 Constructors

[`expected.object.cons`]

- ¹ The exposition-only variable template *`converts-from-any-cvref`* defined in [`optional.ctor`] is used by some constructors for `expected`.

```
constexpr expected();
```

- ² *Constraints:* `is_default_constructible_v<T>` is `true`.

- ³ *Effects:* Value-initializes `val`.

- ⁴ *Postconditions:* `has_value()` is `true`.

- ⁵ *Throws:* Any exception thrown by the initialization of `val`.

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

- ⁶ *Effects:* If `rhs.has_value()` is `true`, direct-non-list-initializes `val` with `*rhs`. Otherwise, direct-non-list-initializes `unex` with `rhs.error()`.

- ⁷ *Postconditions:* `rhs.has_value() == this->has_value()`.

- ⁸ *Throws:* Any exception thrown by the initialization of `val` or `unex`.

- ⁹ *Remarks:* This constructor is defined as deleted unless

- (9.1) — `is_copy_constructible_v<T>` is `true` and

- (9.2) — `is_copy_constructible_v<E>` is `true` [or `is\_reference\_v<E>` is `true`](#).

- ¹⁰ This constructor is trivial if

- (10.1) — `is_trivially_copy_constructible_v<T>` is `true` and

- (10.2) — `is_trivially_copy_constructible_v<E>` is `true`.

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

- ¹¹ *Constraints:*

- (11.1) — `is_move_constructible_v<T>` is `true` and

- (11.2) — `is_move_constructible_v<E>` is `true` [or `is\_reference\_v<E>` is `true`](#).

- ¹² *Effects:* If `rhs.has_value()` is `true`, direct-non-list-initializes `val` with `std::move(*rhs)`. Otherwise, direct-non-list-initializes `unex` with `std::move(rhs.error())`.

- ¹³ *Postconditions:* `rhs.has_value()` is unchanged; `rhs.has_value() == this->has_value()` is `true`.

- ¹⁴ *Throws:* Any exception thrown by the initialization of `val` or `unex`.

- ¹⁵ *Remarks:* The exception specification is equivalent to `is_nothrow_move_constructible_v<T> && is_nothrow_move_constructible_v<E>`.

16 This constructor is trivial if

(16.1) — `is_trivially_move_constructible_v<T>` is true and

(16.2) — `is_trivially_move_constructible_v<E>` is true.

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

17 Let:

(17.1) — `UF` be `const U&` for the first overload and `U` for the second overload.

(17.2) — `GF` be `const G&` for the first overload and `G` for the second overload.

18 *Constraints:*

(18.1) — `is_constructible_v<T, UF>` is true; and

(18.2) — `is_constructible_v<E, GF>` is true; and

(18.3) — if `T` is not `cv bool`, `converts-from-any-cvref<T, expected<U, G>>` is false; and

(18.4) — `is_constructible_v<unexpected<E>, expected<U, G>&>` is false; and

(18.5) — `is_constructible_v<unexpected<E>, expected<U, G>>` is false; and

(18.6) — `is_constructible_v<unexpected<E>, const expected<U, G>&>` is false; and

(18.7) — `is_constructible_v<unexpected<E>, const expected<U, G>>` is false.

19 *Effects:* If `rhs.has_value()`, direct-non-list-initializes `val` with `std::forward<UF>(*rhs)`. Otherwise, direct-non-list-initializes `unex` with `std::forward<GF>(rhs.error())`.

20 *Postconditions:* `rhs.has_value()` is unchanged; `rhs.has_value() == this->has_value()` is true.

21 *Throws:* Any exception thrown by the initialization of `val` or `unex`.

22 *Remarks:* The expression inside `explicit` is equivalent to `!is_convertible_v<UF, T> || !is_convertible_v<GF, E>`.

```
template<class U = remove_cv_t<T>>
constexpr explicit(!is_convertible_v<U, T>) expected(U&& v);
```

23 *Constraints:*

(23.1) — `is_same_v<remove_cvref_t<U>, in_place_t>` is false; and

(23.2) — `is_same_v<remove_cvref_t<U>, expected>` is false; and

(23.3) — `is_same_v<remove_cvref_t<U>, unexpected_t>` is false; and

(23.4) — `remove_cvref_t<U>` is not a specialization of `unexpected`; and

(23.5) — `is_constructible_v<T, U>` is true; and

(23.6) — if `T` is `cv bool`, `remove_cvref_t<U>` is not a specialization of `expected`.

24 *Effects:* Direct-non-list-initializes `val` with `std::forward<U>(v)`.

25 *Postconditions:* `has_value()` is true.

26 *Throws:* Any exception thrown by the initialization of `val`.

```
template<class G>
constexpr explicit(!is_convertible_v<const G&, E>) expected(const unexpected<G>& e);
template<class G>
constexpr explicit(!is_convertible_v<G, E>) expected(unexpected<G>&& e);
```

27 Let `GF` be `const G&` for the first overload and `G` for the second overload.

28 *Constraints:* `is_constructible_v<E, GF>` is true.

29 *Effects:* Direct-non-list-initializes `unex` with `std::forward<GF>(e.error())`.

30 *Postconditions:* `has_value()` is false.

31 *Throws:* Any exception thrown by the initialization of `unex`.

```

template<class... Args>
constexpr explicit expected(in_place_t, Args&&... args);
32     Constraints: is_constructible_v<T, Args...> is true.
33     Effects: Direct-non-list-initializes val with std::forward<Args>(args)....
34     Postconditions: has_value() is true.
35     Throws: Any exception thrown by the initialization of val.

template<class U, class... Args>
constexpr explicit expected(in_place_t, initializer_list<U> il, Args&&... args);
36     Constraints: is_constructible_v<T, initializer_list<U>&, Args...> is true.
37     Effects: Direct-non-list-initializes val with il, std::forward<Args>(args)....
38     Postconditions: has_value() is true.
39     Throws: Any exception thrown by the initialization of val.

template<class... Args>
constexpr explicit expected(unexpect_t, Args&&... args);
40     Constraints: is_constructible_v<E, Args...> is true.
41     Effects: Direct-non-list-initializes unex with std::forward<Args>(args)....
42     Postconditions: has_value() is false.
43     Throws: Any exception thrown by the initialization of unex.

template<class U, class... Args>
constexpr explicit expected(unexpect_t, initializer_list<U> il, Args&&... args);
44     Constraints: is_reference_v<E> is false, and is_constructible_v<E, initializer_list<U>&,
    Args...> is true.
45     Effects: Direct-non-list-initializes unex with il, std::forward<Args>(args)....
46     Postconditions: has_value() is false.
47     Throws: Any exception thrown by the initialization of unex.
48     Remarks: An overload with the same parameter types is defined as deleted when E is an lvalue reference
    type. An initializer list cannot provide the required long-lived error referent.

```

8.1.6.3 Destructor

[expected.object.dtor]

```

constexpr ~expected();
1     Effects: If has_value() is true, destroys val, otherwise destroys unex.
2     Remarks: If is_trivially_destructible_v<T> is true, and is_trivially_destructible_v<E> is
    true, then this destructor is a trivial destructor.

```

8.1.6.4 Assignment

[expected.object.assign]

¹ This subclause makes use of the following exposition-only function template:

```

template<class T, class U, class... Args>
constexpr void reinit-expected(T& newval, U& oldval, Args&&... args) { // exposition only
    if constexpr (is_nothrow_constructible_v<T, Args...>) {
        destroy_at(addressof(oldval));
        construct_at(addressof(newval), std::forward<Args>(args)...);
    } else if constexpr (is_nothrow_move_constructible_v<T>) {
        T tmp(std::forward<Args>(args)...);
        destroy_at(addressof(oldval));
        construct_at(addressof(newval), std::move(tmp));
    } else {
        U tmp(std::move(oldval));
        destroy_at(addressof(oldval));
        try {
            construct_at(addressof(newval), std::forward<Args>(args)...);
        }
    }
}

```

```

    } catch (...) {
        construct_at(addressof(oldval), std::move(tmp));
        throw;
    }
}
}

```

```
constexpr expected& operator=(const expected& rhs);
```

2 *Effects:*

(2.1) — If `this->has_value()` && `rhs.has_value()` is true, equivalent to `val = *rhs`.

(2.2) — Otherwise, if `this->has_value()` is true, equivalent to:

```
reinit-expected(unex, val, rhs.error())
```

(2.3) — Otherwise, if `rhs.has_value()` is true, equivalent to:

```
reinit-expected(val, unex, *rhs)
```

(2.4) — Otherwise, equivalent to `unex = rhs.unex`.

Then, if no exception was thrown, equivalent to: `has_val = rhs.has_value(); return *this;`

3 When `E` is an lvalue reference type, each of the cases above that initializes or assigns `unex` rebinds it: `unex` comes to refer to the same object as `rhs`'s error. No previously or subsequently referenced object is assigned through.

4 *Returns:* `*this`.

5 *Remarks:* This operator is defined as deleted unless:

(5.1) — `is_copy_assignable_v<T>` is true and

(5.2) — `is_copy_constructible_v<T>` is true and

(5.3) — `is_copy_assignable_v<E>` is true or `is_reference_v<E>` is true and

(5.4) — `is_copy_constructible_v<E>` is true or `is_reference_v<E>` is true and

(5.5) — `is_nothrow_move_constructible_v<T> || is_nothrow_move_constructible_v<E>` is true.

6 This operator is trivial if:

(6.1) — `is_trivially_copy_constructible_v<T>` is true, and

(6.2) — `is_trivially_copy_assignable_v<T>` is true, and

(6.3) — `is_trivially_destructible_v<T>` is true, and

(6.4) — `is_trivially_copy_constructible_v<E>` is true, and

(6.5) — `is_trivially_copy_assignable_v<E>` is true, and

(6.6) — `is_trivially_destructible_v<E>` is true.

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

7 *Constraints:*

(7.1) — `is_move_constructible_v<T>` is true and

(7.2) — `is_move_assignable_v<T>` is true and

(7.3) — `is_move_constructible_v<E>` is true or `is_reference_v<E>` is true and

(7.4) — `is_move_assignable_v<E>` is true or `is_reference_v<E>` is true and

(7.5) — `is_nothrow_move_constructible_v<T> || is_nothrow_move_constructible_v<E>` is true.

8 *Effects:*

(8.1) — If `this->has_value()` && `rhs.has_value()` is true, equivalent to `val = std::move(*rhs)`.

(8.2) — Otherwise, if `this->has_value()` is true, equivalent to:

```
reinit-expected(unex, val, std::move(rhs.error()))
```

(8.3) — Otherwise, if `rhs.has_value()` is true, equivalent to:

```
reinit-expected(val, unex, std::move(*rhs))
```

(8.4) — Otherwise, equivalent to `unex = std::move(rhs.unex)`.

Then, if no exception was thrown, equivalent to: `has_val = rhs.has_value(); return *this;`

9 When E is an lvalue reference type, each of the cases above that initializes or assigns `unex` rebinds it: `unex` comes to refer to the same object as `rhs`'s error. No previously or subsequently referenced object is assigned through.

10 *Returns:* `*this`.

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

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

12 This operator is trivial if:

(12.1) — `is_trivially_move_constructible_v<T>` is true, and

(12.2) — `is_trivially_move_assignable_v<T>` is true, and

(12.3) — `is_trivially_destructible_v<T>` is true, and

(12.4) — `is_trivially_move_constructible_v<E>` is true, and

(12.5) — `is_trivially_move_assignable_v<E>` is true, and

(12.6) — `is_trivially_destructible_v<E>` is true.

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

13 *Constraints:*

(13.1) — `is_same_v<expected, remove_cvref_t<U>>` is false; and

(13.2) — `remove_cvref_t<U>` is not a specialization of `unexpected`; and

(13.3) — `is_constructible_v<T, U>` is true; and

(13.4) — `is_assignable_v<T&, U>` is true; and

(13.5) — `is_nothrow_constructible_v<T, U> || is_nothrow_move_constructible_v<T> ||
is_nothrow_move_constructible_v<E>` is true.

14 *Effects:*

(14.1) — If `has_value()` is true, equivalent to: `val = std::forward<U>(v);`

(14.2) — Otherwise, equivalent to:

```
reinit-expected(val, unex, std::forward<U>(v));  
has_val = true;
```

15 *Returns:* `*this`.

```
template<class G>  
constexpr expected& operator=(const unexpected<G>& e);  
template<class G>  
constexpr expected& operator=(unexpected<G>&& e);
```

16 Let GF be `const G&` for the first overload and `G` for the second overload.

17 *Constraints:*

(17.1) — `is_constructible_v<E, GF>` is true; and

(17.2) — `is_assignable_v<E&, GF>` is true; and

(17.3) — `is_nothrow_constructible_v<E, GF> || is_nothrow_move_constructible_v<T> ||
is_nothrow_move_constructible_v<E>` is true.

18 *Effects:*

(18.1) — If `has_value()` is true, equivalent to:

```
reinit-expected(unex, val, std::forward<GF>(e.error()));  
has_val = false;
```

(18.2) — Otherwise, equivalent to: `unex = unexpected<E>(std::forward<GF>(e.error()));`

19 *Returns: *this.*

```

template<class... Args>
constexpr T& emplace(Args&&... args) noexcept;
20 Constraints: is_nothrow_constructible_v<T, Args...> is true.
21 Effects: Equivalent to:
    if (has_value()) {
        destroy_at(addressof(val));
    } else {
        destroy_at(addressof(unex));
        has_val = true;
    }
    return *construct_at(addressof(val), std::forward<Args>(args)...);

template<class U, class... Args>
constexpr T& emplace(initializer_list<U> il, Args&&... args) noexcept;
22 Constraints: is_nothrow_constructible_v<T, initializer_list<U>&, Args...> is true.
23 Effects: Equivalent to:
    if (has_value()) {
        destroy_at(addressof(val));
    } else {
        destroy_at(addressof(unex));
        has_val = true;
    }
    return *construct_at(addressof(val), il, std::forward<Args>(args)...);

```

8.1.6.5 Swap

[expected.object.swap]

```

constexpr void swap(expected& rhs) noexcept(see below);
1 Constraints:
(1.1) — is_swappable_v<T> is true and
(1.2) — is_swappable_v<E> is true and
(1.3) — is_move_constructible_v<T> && is_move_constructible_v<E> is true, and
(1.4) — is_nothrow_move_constructible_v<T> || is_nothrow_move_constructible_v<E> is true.
2 Effects: See .

```

Table 1 — swap(expected&) effects [tab:expected.object.swap]

	this->has_value()	!this->has_value()
rhs.has_value()	equivalent to: using std::swap; swap(val, rhs.val);	calls rhs.swap(*this)
!rhs.has_value()	see below	equivalent to: using std::swap; swap(unex, rhs.unex);

For the case where rhs.has_value() is false and this->has_value() is true, equivalent to:

```

if constexpr (is_nothrow_move_constructible_v<unexpected<E>>) {
    unexpected<E> tmp(std::move(rhs.unex));
    destroy_at(addressof(rhs.unex));
    try {
        construct_at(addressof(rhs.val), std::move(val));
        destroy_at(addressof(val));
        construct_at(addressof(unex), std::move(tmp));
    } catch(...) {
        construct_at(addressof(rhs.unex), std::move(tmp));
        throw;
    }
} else {

```

```

    remove_cv_t<T> tmp(std::move(val));
    destroy_at(addressof(val));
    try {
        construct_at(addressof(uneq), std::move(rhs.uneq));
        destroy_at(addressof(rhs.uneq));
        construct_at(addressof(rhs.val), std::move(tmp));
    } catch (...) {
        construct_at(addressof(val), std::move(tmp));
        throw;
    }
}
has_val = false;
rhs.has_val = true;

```

3 *Throws:* Any exception thrown by the expressions in the *Effects*.

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

```

    is_nothrow_move_constructible_v<T> && is_nothrow_swappable_v<T> &&
    is_nothrow_move_constructible_v<E> && (is_reference_v<E> || is_nothrow_swappable_v<E>)

```

```

friend constexpr void swap(expected& x, expected& y) noexcept(noexcept(x.swap(y)));

```

5 *Effects:* Equivalent to `x.swap(y)`.

8.1.6.6 Observers

[expected.object.obs]

```

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

```

1 *Hardened preconditions:* `has_value()` is true.

2 *Returns:* `addressof(val)`.

```

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

```

3 *Hardened preconditions:* `has_value()` is true.

4 *Returns:* `val`.

```

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

```

5 *Hardened preconditions:* `has_value()` is true.

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

```

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

```

7 *Returns:* `has_val`.

```

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

```

8 *Mandates:* `is_copy_constructible_v<E>` is true.

9 *Returns:* `val`, if `has_value()` is true.

10 *Throws:* `bad_expected_access(as_const(error()))` if `has_value()` is false.

```

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

```

11 *Mandates:* `is_copy_constructible_v<E>` is true and `is_constructible_v<E, decltype(std::move(error()))>` is true.

12 *Returns:* `std::move(val)`, if `has_value()` is true.

13 *Throws:* `bad_expected_access(std::move(error()))` if `has_value()` is false.

```

constexpr const E& error() const & noexcept;

```

```

constexpr E& error() & noexcept;
14   Hardened preconditions: has_value() is false.
15   Returns: unex.error().

constexpr E&& error() && noexcept;
constexpr const E&& error() const && noexcept;
16   Hardened preconditions: has_value() is false.
17   Returns: std::move(unex).error().

template<class U = remove_cv_t<T>> constexpr T value_or(U&& v) const &;
18   Mandates: is_copy_constructible_v<T> is true and is_convertible_v<U, T> is true.
19   Returns: has_value() ? **this : static_cast<T>(std::forward<U>(v)).

template<class U = remove_cv_t<T>> constexpr T value_or(U&& v) &&;
20   Mandates: is_move_constructible_v<T> is true and is_convertible_v<U, T> is true.
21   Returns: has_value() ? std::move(**this) : static_cast<T>(std::forward<U>(v)).

template<class G = remove_cv_t<remove_reference_t<E>>>
constexpr remove_cv_t<remove_reference_t<E>> error_or(G&& e) const &;
22   Mandates: is_copy_constructible_v<remove_cv_t<remove_reference_t<E>>> is true and is_
convertible_v<G, remove_cv_t<remove_reference_t<E>>> is true.
23   Returns: std::forward<G>(e) if has_value() is true, error() otherwise.

template<class G = remove_cv_t<remove_reference_t<E>>>
constexpr remove_cv_t<remove_reference_t<E>> error_or(G&& e) &&;
24   Mandates: is_move_constructible_v<remove_cv_t<remove_reference_t<E>>> is true and is_
convertible_v<G, remove_cv_t<remove_reference_t<E>>> is true.
25   Returns: std::forward<G>(e) if has_value() is true, std::move(error()) otherwise.

```

8.1.6.7 Monadic operations

[expected.object.monadic]

```

template<class F> constexpr auto and_then(F&& f) &;
template<class F> constexpr auto and_then(F&& f) const &;
1   Let U be remove_cvref_t<invoke_result_t<F, decltype((val))>>>.
2   Constraints: is_constructible_v<E, decltype(error())> is true.
3   Mandates: U is a specialization of expected and is_same_v<typename U::error_type, E> is true.
4   Effects: Equivalent to:
        if (has_value())
            return invoke(std::forward<F>(f), val);
        else
            return U(unexpect, error());

template<class F> constexpr auto and_then(F&& f) &&;
template<class F> constexpr auto and_then(F&& f) const &&;
5   Let U be remove_cvref_t<invoke_result_t<F, decltype(std::move(val))>>>.
6   Constraints: is_constructible_v<E, decltype(std::move(error()))> is true.
7   Mandates: U is a specialization of expected and is_same_v<typename U::error_type, E> is true.
8   Effects: Equivalent to:
        if (has_value())
            return invoke(std::forward<F>(f), std::move(val));
        else
            return U(unexpect, std::move(error()));

```

```

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

9   Let G be remove_cvref_t<invoke_result_t<F, decltype(error())>>.
10  Constraints: is_constructible_v<T, decltype((val))> is true.
11  Mandates: G is a specialization of expected and is_same_v<typename G::value_type, T> is true.
12  Effects: Equivalent to:
      if (has_value())
          return G(in_place, val);
      else
          return invoke(std::forward<F>(f), error());

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

13  Let G be remove_cvref_t<invoke_result_t<F, decltype(std::move(error()))>>.
14  Constraints: is_constructible_v<T, decltype(std::move(val))> is true.
15  Mandates: G is a specialization of expected and is_same_v<typename G::value_type, T> is true.
16  Effects: Equivalent to:
      if (has_value())
          return G(in_place, std::move(val));
      else
          return invoke(std::forward<F>(f), std::move(error()));

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

17  Let U be remove_cv_t<invoke_result_t<F, decltype((val))>>.
18  Constraints: is_constructible_v<E, decltype(error())> is true.
19  Mandates: U is a valid value type for expected. If is_void_v<U> is false, the declaration
      U u(invoke(std::forward<F>(f), val));
is well-formed.

20  Effects:
(20.1) — If has_value() is false, returns expected<U, E>(unexpected, error()).
(20.2) — Otherwise, if is_void_v<U> is false, returns an expected<U, E> object whose has_val member
is true and val member is direct-non-list-initialized with invoke(std::forward<F>(f), val).
(20.3) — Otherwise, evaluates invoke(std::forward<F>(f), val) and then returns expected<U, E>().

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

21  Let U be remove_cv_t<invoke_result_t<F, decltype(std::move(val))>>.
22  Constraints: is_constructible_v<E, decltype(std::move(error()))> is true.
23  Mandates: U is a valid value type for expected. If is_void_v<U> is false, the declaration
      U u(invoke(std::forward<F>(f), std::move(val)));
is well-formed.

24  Effects:
(24.1) — If has_value() is false, returns expected<U, E>(unexpected, std::move(error())).
(24.2) — Otherwise, if is_void_v<U> is false, returns an expected<U, E> object whose has_val mem-
ber is true and val member is direct-non-list-initialized with invoke(std::forward<F>(f),
std::move(val)).
(24.3) — Otherwise, evaluates invoke(std::forward<F>(f), std::move(val)) and then returns ex-
pected<U, E>().

```

```

template<class F> constexpr auto transform_error(F&& f) &;
template<class F> constexpr auto transform_error(F&& f) const &;
25     Let G be remove_cv_t<invoke_result_t<F, decltype(error())>>.
26     Constraints: is_constructible_v<T, decltype((val))> is true.
27     Mandates: G is a valid template argument for unexpected and the declaration
        G g(invoke(std::forward<F>(f), error()));
        is well-formed.
28     Returns: If has_value() is true, expected<T, G>(in_place, val); otherwise, an expected<T,
        G> object whose has_val member is false and unex member is direct-non-list-initialized with
        invoke(std::forward<F>(f), error()).

template<class F> constexpr auto transform_error(F&& f) &&;
template<class F> constexpr auto transform_error(F&& f) const &&;
29     Let G be remove_cv_t<invoke_result_t<F, decltype(std::move(error()))>>.
30     Constraints: is_constructible_v<T, decltype(std::move(val))> is true.
31     Mandates: G is a valid template argument for unexpected and the declaration
        G g(invoke(std::forward<F>(f), std::move(error())));
        is well-formed.
32     Returns: If has_value() is true, expected<T, G>(in_place, std::move(val)); otherwise, an
        expected<T, G> object whose has_val member is false and unex member is direct-non-list-initialized
        with invoke(std::forward<F>(f), std::move(error())).

```

8.1.6.8 Equality operators

[expected.object.eq]

```

template<class T2, class E2> requires (!is_void_v<T2>)
    friend constexpr bool operator==(const expected& x, const expected<T2, E2>& y);
1     Constraints: The expressions *x == *y and x.error() == y.error() are well-formed and their results
        are convertible to bool.
2     Returns: If x.has_value() does not equal y.has_value(), false; otherwise if x.has_value() is true,
        *x == *y; otherwise x.error() == y.error().

template<class T2> friend constexpr bool operator==(const expected& x, const T2& v);
3     Constraints: T2 is not a specialization of expected. The expression *x == v is well-formed and its
        result is convertible to bool.
        [Note 1: T need not be Cpp17EqualityComparable. — end note]
4     Returns: If x.has_value() is true, *x == v; otherwise false.

template<class E2> friend constexpr bool operator==(const expected& x, const unexpected<E2>& e);
5     Constraints: The expression x.error() == e.error() is well-formed and its result is convertible to
        bool.
6     Returns: If !x.has_value() is true, x.error() == e.error(); otherwise false.

```

8.1.7 Partial specialization of expected for void types

[expected.void]

8.1.7.1 General

[expected.void.general]

```

template<class T, class E> requires is_void_v<T>
class expected<T, E> {
public:
    using value_type = T;
    using error_type = E;
    using unexpected_type = unexpected<E>;

    template<class U>
    using rebind = expected<U, error_type>;

```

```

// 8.1.7.2, constructors
constexpr expected() noexcept;
constexpr expected(const expected&);
constexpr expected(expected&&) noexcept(see below);
template<class U, class G>
    constexpr explicit(see below) expected(const expected<U, G>&);
template<class U, class G>
    constexpr explicit(see below) expected(expected<U, G>&&);

template<class G>
    constexpr explicit(see below) expected(const unexpected<G>&);
template<class G>
    constexpr explicit(see below) expected(unexpected<G>&&);

constexpr explicit expected(in_place_t) noexcept;
template<class... Args>
    constexpr explicit expected(unexpect_t, Args&&...);
template<class U, class... Args>
    constexpr explicit expected(unexpect_t, initializer_list<U>, Args&&...);

// 8.1.7.3, destructor
constexpr ~expected();

// 8.1.7.4, assignment
constexpr expected& operator=(const expected&);
constexpr expected& operator=(expected&&) noexcept(see below);
template<class G>
    constexpr expected& operator=(const unexpected<G>&);
template<class G>
    constexpr expected& operator=(unexpected<G>&&);
constexpr void emplace() noexcept;

// 8.1.7.5, swap
constexpr void swap(expected&) noexcept(see below);
friend constexpr void swap(expected& x, expected& y) noexcept(noexcept(x.swap(y)));

// 8.1.7.6, observers
constexpr explicit operator bool() const noexcept;
constexpr bool has_value() const noexcept;
constexpr void operator*() const noexcept;
constexpr void value() const &; // freestanding-deleted
constexpr void value() &&; // freestanding-deleted
constexpr const E& error() const & noexcept;
constexpr E& error() & noexcept;
constexpr const E&& error() const && noexcept;
constexpr E&& error() && noexcept;
template<class G = remove_cv_t<remove_reference_t<E>>>
    constexpr remove_cv_t<remove_reference_t<E>> error_or(G&&) const &;
template<class G = remove_cv_t<remove_reference_t<E>>>
    constexpr remove_cv_t<remove_reference_t<E>> error_or(G&&) &&;

// 8.1.7.7, 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 or_else(F&& f) &;
template<class F> constexpr auto or_else(F&& f) &&;
template<class F> constexpr auto or_else(F&& f) const &;
template<class F> constexpr auto or_else(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 auto transform_error(F&& f) &;
template<class F> constexpr auto transform_error(F&& f) &&;
template<class F> constexpr auto transform_error(F&& f) const &;
template<class F> constexpr auto transform_error(F&& f) const &&;

// 8.1.7.8, equality operators
template<class T2, class E2> requires is_void_v<T2>
    friend constexpr bool operator==(const expected& x, const expected<T2, E2>& y);
template<class E2>
    friend constexpr bool operator==(const expected&, const unexpected<E2>&);

private:
    bool has_val;           // exposition only
    union {
        unexpected<E> unex; // exposition only
    };
};

```

- ¹ Any object of type `expected<T, E>` either represents a value of type `T`, or contains a value of type `E` nested within it. Member `has_val` indicates whether the `expected<T, E>` object represents a value of type `T`.
- ² When `has_value()` is false, the error is `unex.error()`. The error is held as an `unexpected<E>`, and not as an `E`, so that `E` may be an lvalue reference type: an `E&` cannot be a union member, whereas `unexpected<E&>` holds a pointer to an external object.
- ³ A program that instantiates the definition of the template `expected<T, E>` with a type for the `E` parameter that is not a valid template argument for `unexpected` is ill-formed.
- ⁴ If `E` is not an lvalue reference type, `E` shall meet the requirements of *Cpp17Destructible* ().

8.1.7.2 Constructors

[expected.void.cons]

```
constexpr expected() noexcept;
```

- ¹ *Postconditions:* `has_value()` is true.

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

- ² *Effects:* If `rhs.has_value()` is false, direct-non-list-initializes `unex` with `rhs.error()`.
- ³ *Postconditions:* `rhs.has_value() == this->has_value()`.
- ⁴ *Throws:* Any exception thrown by the initialization of `unex`.
- ⁵ *Remarks:* This constructor is defined as deleted unless `is_copy_constructible_v<E>` is true or is_reference_v<E> is true.
- ⁶ This constructor is trivial if `is_trivially_copy_constructible_v<E>` is true.

```
constexpr expected(expected&& rhs) noexcept(is_nothrow_move_constructible_v<E>);
```

- ⁷ *Constraints:* `is_move_constructible_v<E>` is true or is_reference_v<E> is true.
- ⁸ *Effects:* If `rhs.has_value()` is false, direct-non-list-initializes `unex` with `std::move(rhs.error())`.
- ⁹ *Postconditions:* `rhs.has_value()` is unchanged; `rhs.has_value() == this->has_value()` is true.
- ¹⁰ *Throws:* Any exception thrown by the initialization of `unex`.
- ¹¹ *Remarks:* This constructor is trivial if `is_trivially_move_constructible_v<E>` is true.

```

template<class U, class G>
    constexpr explicit(!is_convertible_v<const G&, E>) expected(const expected<U, G>& rhs);
template<class U, class G>
    constexpr explicit(!is_convertible_v<G, E>) expected(expected<U, G>&& rhs);

```

- ¹² Let `GF` be `const G&` for the first overload and `G` for the second overload.

- ¹³ *Constraints:*

- (13.1) — `is_void_v<U>` is true; and
- (13.2) — `is_constructible_v<E, GF>` is true; and
- (13.3) — `is_constructible_v<unexpected<E>, expected<U, G>&>` is false; and

(13.4) — `is_constructible_v<unexpected<E>, expected<U, G>>` is false; and

(13.5) — `is_constructible_v<unexpected<E>, const expected<U, G>&>` is false; and

(13.6) — `is_constructible_v<unexpected<E>, const expected<U, G>>` is false.

14 *Effects:* If `rhs.has_value()` is false, direct-non-list-initializes `unex` with `std::forward<GF>(rhs.error())`.

15 *Postconditions:* `rhs.has_value()` is unchanged; `rhs.has_value() == this->has_value()` is true.

16 *Throws:* Any exception thrown by the initialization of `unex`.

```
template<class G>
constexpr explicit(!is_convertible_v<const G&, E>) expected(const unexpected<G>& e);
template<class G>
constexpr explicit(!is_convertible_v<G, E>) expected(unexpected<G>&& e);
```

17 Let `GF` be `const G&` for the first overload and `G` for the second overload.

18 *Constraints:* `is_constructible_v<E, GF>` is true.

19 *Effects:* Direct-non-list-initializes `unex` with `std::forward<GF>(e.error())`.

20 *Postconditions:* `has_value()` is false.

21 *Throws:* Any exception thrown by the initialization of `unex`.

```
constexpr explicit expected(in_place_t) noexcept;
```

22 *Postconditions:* `has_value()` is true.

```
template<class... Args>
constexpr explicit expected(unexpect_t, Args&&... args);
```

23 *Constraints:* `is_constructible_v<E, Args...>` is true.

24 *Effects:* Direct-non-list-initializes `unex` with `std::forward<Args>(args)...`

25 *Postconditions:* `has_value()` is false.

26 *Throws:* Any exception thrown by the initialization of `unex`.

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

27 *Constraints:* `is_reference_v<E>` is false, and `is_constructible_v<E, initializer_list<U>&, Args...>` is true.

28 *Effects:* Direct-non-list-initializes `unex` with `il, std::forward<Args>(args)...`

29 *Postconditions:* `has_value()` is false.

30 *Throws:* Any exception thrown by the initialization of `unex`.

31 *Remarks:* An overload with the same parameter types is defined as deleted when `E` is an lvalue reference type. An initializer list cannot provide the required long-lived error referent.

8.1.7.3 Destructor

[expected.void.dtor]

```
constexpr ~expected();
```

1 *Effects:* If `has_value()` is false, destroys `unex`.

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

8.1.7.4 Assignment

[expected.void.assign]

```
constexpr expected& operator=(const expected& rhs);
```

1 *Effects:*

(1.1) — If `this->has_value() && rhs.has_value()` is true, no effects.

(1.2) — Otherwise, if `this->has_value()` is true, equivalent to: `construct_at(addressof(unex), rhs.unex); has_val = false;`

(1.3) — Otherwise, if `rhs.has_value()` is true, destroys `unex` and sets `has_val` to true.

(1.4) — Otherwise, equivalent to `unex = rhs.unex`.

2 When E is an lvalue reference type, each of the cases above that initializes or assigns `unex` rebinds it: `unex` comes to refer to the same object as `rhs`'s error. No previously or subsequently referenced object is assigned through.

3 *Returns:* `*this`.

4 *Remarks:* This operator is defined as deleted unless

(4.1) — `is_copy_assignable_v<E>` is true or `is_reference_v<E>` is true and

(4.2) — `is_copy_constructible_v<E>` is true or `is_reference_v<E>` is true.

5 This operator is trivial if `is_trivially_copy_constructible_v<E>`, `is_trivially_copy_assignable_v<E>`, and `is_trivially_destructible_v<E>` are all true.

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

6 *Constraints:*

(6.1) — `is_move_constructible_v<E>` is true or `is_reference_v<E>` is true and

(6.2) — `is_move_assignable_v<E>` is true or `is_reference_v<E>` is true.

7 *Effects:*

(7.1) — If `this->has_value()` && `rhs.has_value()` is true, no effects.

(7.2) — Otherwise, if `this->has_value()` is true, equivalent to:
`construct_at(addressof(unex), std::move(rhs.unex));`
`has_val = false;`

(7.3) — Otherwise, if `rhs.has_value()` is true, destroys `unex` and sets `has_val` to true.

(7.4) — Otherwise, equivalent to `unex = std::move(rhs.unex)`.

8 When E is an lvalue reference type, each of the cases above that initializes or assigns `unex` rebinds it: `unex` comes to refer to the same object as `rhs`'s error. No previously or subsequently referenced object is assigned through.

9 *Returns:* `*this`.

10 *Remarks:* The exception specification is equivalent to `is_nothrow_move_constructible_v<E>` && `is_nothrow_move_assignable_v<E>`.

11 This operator is trivial if `is_trivially_move_constructible_v<E>`, `is_trivially_move_assignable_v<E>`, and `is_trivially_destructible_v<E>` are all true.

```
template<class G>
constexpr expected& operator=(const unexpected<G>& e);
template<class G>
constexpr expected& operator=(unexpected<G>&& e);
```

12 Let GF be `const G&` for the first overload and `G` for the second overload.

13 *Constraints:* `is_constructible_v<E, GF>` is true and `is_assignable_v<E&, GF>` is true.

14 *Effects:*

(14.1) — If `has_value()` is true, equivalent to:

```
construct_at(addressof(unex), std::forward<GF>(e.error()));
has_val = false;
```

(14.2) — Otherwise, equivalent to: `unex = unexpected<E>(std::forward<GF>(e.error()));`

15 *Returns:* `*this`.

```
constexpr void emplace() noexcept;
```

16 *Effects:* If `has_value()` is false, destroys `unex` and sets `has_val` to true.

8.1.7.5 Swap

[`expected.void.swap`]

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

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

2 *Effects:* See .

Table 2 — `swap(expected&)` effects [tab:expected.void.swap]

	<code>this->has_value()</code>	<code>!this->has_value()</code>
<code>rhs.has_value()</code>	no effects	calls <code>rhs.swap(*this)</code>
<code>!rhs.has_value()</code>	see below	equivalent to: using <code>std::swap</code> ; <code>swap(unex, rhs.unex);</code>

For the case where `rhs.has_value()` is false and `this->has_value()` is true, equivalent to:

```
construct_at(addressof(unex), std::move(rhs.unex));
destroy_at(addressof(rhs.unex));
has_val = false;
rhs.has_val = true;
```

3 *Throws:* Any exception thrown by the expressions in the *Effects*.

4 *Remarks:* The exception specification is equivalent to `is_nothrow_move_constructible_v<E> && (is_reference_v<E> || is_nothrow_swappable_v<E>)`.

```
friend constexpr void swap(expected& x, expected& y) noexcept(noexcept(x.swap(y)));
```

5 *Effects:* Equivalent to `x.swap(y)`.

8.1.7.6 Observers

[expected.void.obs]

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

1 *Returns:* `has_val`.

```
constexpr void operator*() const noexcept;
```

2 *Hardened preconditions:* `has_value()` is true.

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

3 *Mandates:* `is_copy_constructible_v<E>` is true.

4 *Throws:* `bad_expected_access(error())` if `has_value()` is false.

```
constexpr void value() &&;
```

5 *Mandates:* `is_copy_constructible_v<E>` is true and `is_move_constructible_v<E>` is true.

6 *Throws:* `bad_expected_access(std::move(error()))` if `has_value()` is false.

```
constexpr const E& error() const & noexcept;
constexpr E& error() & noexcept;
```

7 *Hardened preconditions:* `has_value()` is false.

8 *Returns:* `unex.error()`.

```
constexpr E&& error() && noexcept;
constexpr const E&& error() const && noexcept;
```

9 *Hardened preconditions:* `has_value()` is false.

10 *Returns:* `std::move(unex).error()`.

```
template<class G = remove_cv_t<remove_reference_t<E>>>
constexpr remove_cv_t<remove_reference_t<E>> error_or(G&& e) const &;
```

11 *Mandates:* `is_copy_constructible_v<remove_cv_t<remove_reference_t<E>>>` is true and `is_convertible_v<G, remove_cv_t<remove_reference_t<E>>>` is true.

12 *Returns:* `std::forward<G>(e)` if `has_value()` is true, `error()` otherwise.

```
template<class G = remove_cv_t<remove_reference_t<E>>>
constexpr remove_cv_t<remove_reference_t<E>> error_or(G&& e) &&;
```

13 *Mandates:* `is_move_constructible_v<remove_cv_t<remove_reference_t<E>>>` is true and `is_convertible_v<G, remove_cv_t<remove_reference_t<E>>>` is true.

14 *Returns:* `std::forward<G>(e)` if `has_value()` is true, `std::move(error())` otherwise.

8.1.7.7 Monadic operations

[expected.void.monadic]

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

1 Let U be `remove_cvref_t<invoke_result_t<F>>`.

2 *Constraints:* `is_constructible_v<E, decltype(error())>` is true.

3 *Mandates:* U is a specialization of `expected` and `is_same_v<typename U::error_type, E>` is true.

4 *Effects:* Equivalent to:

```
    if (has_value())
        return invoke(std::forward<F>(f));
    else
        return U(unexpect, error());
```

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

5 Let U be `remove_cvref_t<invoke_result_t<F>>`.

6 *Constraints:* `is_constructible_v<E, decltype(std::move(error()))>` is true.

7 *Mandates:* U is a specialization of `expected` and `is_same_v<typename U::error_type, E>` is true.

8 *Effects:* Equivalent to:

```
    if (has_value())
        return invoke(std::forward<F>(f));
    else
        return U(unexpect, std::move(error()));
```

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

9 Let G be `remove_cvref_t<invoke_result_t<F, decltype(error())>>`.

10 *Mandates:* G is a specialization of `expected` and `is_same_v<typename G::value_type, T>` is true.

11 *Effects:* Equivalent to:

```
    if (has_value())
        return G();
    else
        return invoke(std::forward<F>(f), error());
```

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

12 Let G be `remove_cvref_t<invoke_result_t<F, decltype(std::move(error()))>>`.

13 *Mandates:* G is a specialization of `expected` and `is_same_v<typename G::value_type, T>` is true.

14 *Effects:* Equivalent to:

```
    if (has_value())
        return G();
    else
        return invoke(std::forward<F>(f), std::move(error()));
```

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

15 Let U be `remove_cv_t<invoke_result_t<F>>`.

16 *Constraints:* `is_constructible_v<E, decltype(error())>` is true.

17 *Mandates:* U is a valid value type for `expected`. If `is_void_v<U>` is `false`, the declaration

U u(`invoke(std::forward<F>(f))`);

is well-formed.

18 *Effects:*

(18.1) — If `has_value()` is `false`, returns `expected<U, E>(unexpected, error())`.

(18.2) — Otherwise, if `is_void_v<U>` is `false`, returns an `expected<U, E>` object whose *has_val* member is `true` and *val* member is direct-non-list-initialized with `invoke(std::forward<F>(f))`.

(18.3) — Otherwise, evaluates `invoke(std::forward<F>(f))` and then returns `expected<U, E>()`.

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

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

19 Let U be `remove_cv_t<invoke_result_t<F>>`.

20 *Constraints:* `is_constructible_v<E, decltype(std::move(error()))>` is `true`.

21 *Mandates:* U is a valid value type for `expected`. If `is_void_v<U>` is `false`, the declaration

U u(`invoke(std::forward<F>(f))`);

is well-formed.

22 *Effects:*

(22.1) — If `has_value()` is `false`, returns `expected<U, E>(unexpected, std::move(error()))`.

(22.2) — Otherwise, if `is_void_v<U>` is `false`, returns an `expected<U, E>` object whose *has_val* member is `true` and *val* member is direct-non-list-initialized with `invoke(std::forward<F>(f))`.

(22.3) — Otherwise, evaluates `invoke(std::forward<F>(f))` and then returns `expected<U, E>()`.

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

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

23 Let G be `remove_cv_t<invoke_result_t<F, decltype(error())>>`.

24 *Mandates:* G is a valid template argument for `unexpected` and the declaration

G g(`invoke(std::forward<F>(f), error())`);

is well-formed.

25 *Returns:* If `has_value()` is `true`, `expected<T, G>()`; otherwise, an `expected<T, G>` object whose *has_val* member is `false` and *unex* member is direct-non-list-initialized with `invoke(std::forward<F>(f), error())`.

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

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

26 Let G be `remove_cv_t<invoke_result_t<F, decltype(std::move(error()))>>`.

27 *Mandates:* G is a valid template argument for `unexpected` and the declaration

G g(`invoke(std::forward<F>(f), std::move(error()))`);

is well-formed.

28 *Returns:* If `has_value()` is `true`, `expected<T, G>()`; otherwise, an `expected<T, G>` object whose *has_val* member is `false` and *unex* member is direct-non-list-initialized with `invoke(std::forward<F>(f), std::move(error()))`.

8.1.7.8 Equality operators

[`expected.void.eq`]

```
template<class T2, class E2> requires is_void_v<T2>
```

```
friend constexpr bool operator==(const expected& x, const expected<T2, E2>& y);
```

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

2 *Returns:* If `x.has_value()` does not equal `y.has_value()`, `false`; otherwise if `x.has_value()` is `true`, `true`; otherwise `x.error() == y.error()`.

```
template<class E2>
friend constexpr bool operator==(const expected& x, const unexpected<E2>& e);
```

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

4 *Returns:* If `!x.has_value()` is true, `x.error() == e.error()`; otherwise false.

8.1.8 Partial specialization of `expected` for reference types

[expected.ref]

8.1.8.1 General

[expected.ref.general]

```
template<class T, class E>
class expected<T&, E> {
public:
    using value_type      = T&;
    using error_type      = E;
    using unexpected_type = unexpected<E>;

    template<class U>
    using rebind = expected<U, error_type>;

    // 8.1.8.2, constructors
    constexpr expected() = delete;
    constexpr expected(const expected&);
    constexpr expected(expected&&) noexcept(see below);
    template<class U = T>
        constexpr explicit(see below) expected(U&&);
    template<class U>
        constexpr expected(U&&) = delete;
    template<class U, class G>
        constexpr explicit(see below) expected(const expected<U&, G>&);
    template<class U, class G>
        constexpr explicit(see below) expected(expected<U&, G>&&);
    template<class G>
        constexpr explicit(see below) expected(const unexpected<G>&);
    template<class G>
        constexpr explicit(see below) expected(unexpected<G>&&);
    template<class... Args>
        constexpr expected(in_place_t, Args&&...) = delete;
    template<class... Args>
        constexpr explicit expected(unexpect_t, Args&&...);
    template<class U, class... Args>
        constexpr explicit expected(unexpect_t, initializer_list<U>, Args&&...);

    // 8.1.8.3, destructor
    constexpr ~expected();

    // 8.1.8.4, assignment
    constexpr expected& operator=(const expected&);
    constexpr expected& operator=(expected&&) noexcept(see below);
    template<class U = T>
        constexpr expected& operator=(U&&);
    template<class G>
        constexpr expected& operator=(const unexpected<G>&);
    template<class G>
        constexpr expected& operator=(unexpected<G>&&);
    template<class U = T>
        constexpr T& emplace(U&&) noexcept(see below);

    // 8.1.8.5, swap
    constexpr void swap(expected&) noexcept(see below);
    friend constexpr void swap(expected& x, expected& y) noexcept(noexcept(x.swap(y)));

    // 8.1.8.6, 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 &;
constexpr T& value() &&;
constexpr const E& error() const & noexcept;
constexpr E& error() & noexcept;
constexpr const E&& error() const && noexcept;
constexpr E&& error() && noexcept;
template<class U = remove_cv_t<T>> constexpr remove_cv_t<T> value_or(U&&) const;
template<class G = remove_cv_t<remove_reference_t<E>>>
    constexpr remove_cv_t<remove_reference_t<E>> error_or(G&&) const &;
template<class G = remove_cv_t<remove_reference_t<E>>>
    constexpr remove_cv_t<remove_reference_t<E>> error_or(G&&) &&;

```

// 8.1.8.7, 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 or_else(F&& f) &;
template<class F> constexpr auto or_else(F&& f) &&;
template<class F> constexpr auto or_else(F&& f) const &;
template<class F> constexpr auto or_else(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 auto transform_error(F&& f) &;
template<class F> constexpr auto transform_error(F&& f) &&;
template<class F> constexpr auto transform_error(F&& f) const &;
template<class F> constexpr auto transform_error(F&& f) const &&;

```

// 8.1.8.8, equality operators

```

template<class T2, class E2> requires (!is_void_v<T2>)
    friend constexpr bool operator==(const expected& x, const expected<T2, E2>& y);
template<class T2>
    friend constexpr bool operator==(const expected&, const T2&);
template<class E2>
    friend constexpr bool operator==(const expected&, const unexpected<E2>&);

```

```

private:
    bool has_val;           // exposition only
    union {
        T* val;             // exposition only
        unexpected<E> unex; // exposition only
    };
};

```

- ¹ An object of type `expected<T&, E>` either represents a reference to an object of type `T`, or holds an error. Member `has_val` indicates whether the object represents a reference. When it represents a reference, member `val` points to the referenced object, which is not owned by the `expected` object. Otherwise, the error is `unex.error()`.
- ² The error is held as an `unexpected<E>`, and not as an `E`, so that `E` can be an lvalue reference type: an `E&` cannot be a union member, whereas `unexpected<E&>` holds a pointer to an external object.
- ³ A program that instantiates the definition of `expected<T&, E>` with a type for the `E` parameter that is not a valid template argument for `unexpected` is ill-formed.
- ⁴ `T` shall be an object type that is not an array type. If `E` is not an lvalue reference type, `E` shall meet the *Cpp17Destructible* requirements ().

8.1.8.2 Constructors

[expected.ref.cons]

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

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

1 *Effects:* If `rhs.has_value()` is true, initializes *val* with `rhs.val`, so that `*this` and `rhs` refer to the same object; otherwise, initializes *unex* with `rhs.unex`.

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

3 *Remarks:* The exception specification of the move constructor is equivalent to `is_nothrow_move_constructible_v<E>`. Each of these constructors is trivial if the corresponding constructor of `E` is trivial, and is defined as deleted unless that constructor of `E` is available.

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

4 *Constraints:*

(4.1) — `remove_cvref_t<U>` is not `in_place_t`, `expected`, or a specialization of `unexpected`;

(4.2) — `is_constructible_v<T&, U>` is true; and

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

5 *Effects:* Let `r` be the lvalue result of `T& r = std::forward<U>(u);`. Initializes *val* with `addressof(r)`.

6 *Postconditions:* `has_value()` is true.

7 *Remarks:* The expression inside `explicit` is equivalent to `!is_convertible_v<U, T&>`. A constructor for which `reference_constructs_from_temporary_v<T&, U>` is true — one that would bind `T&` to a temporary — is defined as deleted.

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

8 *Constraints:*

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

(8.2) — `reference_constructs_from_temporary_v<T&, U&>` is false; and

(8.3) — if `E` is an lvalue reference type, `G` is an lvalue reference type and `is_convertible_v<G, E>` is true; otherwise `is_constructible_v<E, const G&>` (for the first overload) or `is_constructible_v<E, G>` (for the second) is true.

9 *Effects:* If `rhs.has_value()` is true, initializes *val* with `addressof(*rhs)`, so that `*this` refers to the object referred to by `rhs`; otherwise, initializes *unex* with the error of `rhs`. No object referred to by `rhs` is moved from.

10 *Remarks:* The expression inside `explicit` is equivalent to `!is_convertible_v<U&, T&> || !is_convertible_v<G, E>`.

```
template<class G>
constexpr explicit(see below) expected(const unexpected<G>& e);
template<class G>
constexpr explicit(see below) expected(unexpected<G>&& e);
```

11 *Constraints:* If `E` is an lvalue reference type, all of the following are true:

(11.1) — `G` is an lvalue reference type,

(11.2) — `is_convertible_v<G, E>` is true, and

(11.3) — `reference_constructs_from_temporary_v<E, G>` is false.

Otherwise, `is_constructible_v<E, const G&>` is true for the first overload, or `is_constructible_v<E, G>` is true for the second overload.

12 *Effects:* Initializes *unex* with the error of `e`. `has_value()` is false.

13 *Remarks:* When `E` is an lvalue reference type and `G` is not a reference type, these constructors are defined as deleted: the referent would be the value owned by `e`, and binding `E` to it would dangle once `e` is destroyed.

```

template<class... Args>
constexpr explicit expected(unexpect_t, Args&&... args);
template<class U, class... Args>
constexpr explicit expected(unexpect_t, initializer_list<U> il, Args&&... args);

```

14 *Constraints:* `is_constructible_v<E, Args...>` (respectively, `is_reference_v<E>` is false and `is_constructible_v<E, initializer_list<U>&, Args...>`) is true. If E is an lvalue reference type, the initialization shall not bind E to a temporary.

15 *Effects:* Direct-non-list-initializes `unex` with `in_place` and `std::forward<Args>(args)...` (respectively, `in_place`, `il`, and `std::forward<Args>(args)...`), so that the error `unex.error()` is constructed from those arguments. `has_value()` is false.

16 *Remarks:* The initializer-list overload is defined as deleted when E is an lvalue reference type. An initializer list cannot provide the required long-lived error referent.

8.1.8.3 Destructor

[expected.ref.dtor]

```
constexpr ~expected();
```

1 *Effects:* If `has_value()` is false, destroys `unex`.

2 *Remarks:* This destructor is trivial if E is trivially destructible. T is not destroyed; `*this` never owns the object it refers to.

8.1.8.4 Assignment

[expected.ref.assign]

1 Assignment rebinds: assigning to an `expected<T&, E>` that holds a value changes which object it refers to. It never assigns through to the referent.

```

constexpr expected& operator=(const expected& rhs);
constexpr expected& operator=(expected&& rhs) noexcept(see below);

```

2 *Effects:* If `rhs.has_value()` is true: if `has_value()` is true, assigns `rhs.val` to `val`; otherwise destroys `unex` and initializes `val` with `rhs.val`. If `rhs.has_value()` is false, the error of `rhs` is assigned to or used to initialize `unex`, as for the primary template (8.1.6.4). In every case `*this` comes to refer to the object `rhs` refers to, or to hold the error of `rhs`; no referenced object is assigned through.

3 *Returns:* `*this`.

```

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

```

4 *Constraints:* All of the following are true:

(4.1) — `remove_cvref_t<U>` is neither `expected` nor a specialization of `unexpected`,

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

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

5 *Effects:* Let `r` be the lvalue result of `T& r = std::forward<U>(u);`. If `has_value()` is true, assigns `addressof(r)` to `val`. Otherwise, destroys `unex`, initializes `val` with `addressof(r)`, and sets `has_val` to true; if binding `r` throws, `*this` is left unchanged.

6 *Returns:* `*this`.

```

template<class G>
constexpr expected& operator=(const unexpected<G>& e);
template<class G>
constexpr expected& operator=(unexpected<G>&& e);

```

7 *Constraints:* If E is an lvalue reference type, G is an lvalue reference type, `is_convertible_v<G, E>` is true, and `reference_constructs_from_temporary_v<E, G>` is false; otherwise the corresponding requirements of the primary template hold.

8 *Effects:* Makes `*this` hold the error of `e`, reinitializing `unex` from `e` rather than assigning through it. When E is a reference type, `unex.error()` thereafter refers to the same object as `e.error()`; the previously referenced object, if any, is not modified.

9 *Returns:* `*this`.

10 *Remarks:* When `E` is an lvalue reference type and `G` is not a reference type, these overloads are defined as deleted.

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

11 *Constraints:* `is_constructible_v<T&, U>` is true and `reference_constructs_from_temporary_v<T&, U>` is false.

12 *Effects:* Rebinds `*this` to refer to the object bound by `T& r = std::forward<U>(u);`; if `has_value()` is false, destroys `unex` first. Sets `val` to `addressof(r)` and `has_val` to true.

13 *Returns:* `*val`.

8.1.8.5 Swap

[expected.ref.swap]

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

1 *Effects:* Exchanges the states of `*this` and `rhs`. When both hold values, exchanges `val` and `rhs.val` — the referenced objects are not swapped. Otherwise behaves as the primary template's `swap` does for the error.

2 *Remarks:* The exception specification is equivalent to `is_nothrow_move_constructible_v<E> && (is_reference_v<E> || is_nothrow_swappable_v<E>)`.

```
friend constexpr void swap(expected& x, expected& y) noexcept(noexcept(x.swap(y)));
```

3 *Effects:* Equivalent to `x.swap(y)`.

8.1.8.6 Observers

[expected.ref.obs]

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

1 *Preconditions:* `has_value()` is true.

2 *Returns:* `val` and `*val` respectively.

3 *Remarks:* These are `const` member functions that return a non-`const` `T*` and `T&`; the constness of `*this` does not propagate to the referenced object. For deep `const`, use `expected<const T&, E>`.

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

4 *Returns:* `has_val`.

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

5 *Returns:* `*val` if `has_value()` is true.

6 *Throws:* For the first overload, `bad_expected_access(as_const(error()))` if `has_value()` is false. For the second overload, `bad_expected_access(std::move(error()))` if `has_value()` is false. Both overloads yield `T&`; the value category of `*this` does not affect the returned reference.

```
constexpr const E& error() const & noexcept;
constexpr E& error() & noexcept;
constexpr const E&& error() const && noexcept;
constexpr E&& error() && noexcept;
```

7 *Preconditions:* `has_value()` is false.

8 *Returns:* `unex.error()`, with the `const` and reference qualification of the selected overload.

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

9 Let `X` be `remove_cv_t<T>`.

10 *Mandates:* `is_convertible_v<T&, X>` and `is_convertible_v<U, X>` are true.

11 *Returns:*

```

has_value()
? static_cast<X>(*val)
: static_cast<X>(std::forward<U>(v))

```

The result is an object, never a reference.

```

template<class G = remove_cv_t<remove_reference_t<E>>>
constexpr remove_cv_t<remove_reference_t<E>> error_or(G&& e) const &;
template<class G = remove_cv_t<remove_reference_t<E>>>
constexpr remove_cv_t<remove_reference_t<E>> error_or(G&& e) &&;

```

- ¹² *Returns:* Each overload behaves as the corresponding overload for the primary template. The result is an object, never a reference.

8.1.8.7 Monadic operations

[expected.ref.monadic]

- ¹ The member templates `and_then`, `or_else`, `transform`, and `transform_error` behave as specified for the primary template (8.1.6.7), with one difference: the value is passed to the callable as `T&` for every ref-qualification of `*this`. An rvalue `expected<T&, E>` does not pass its referent as an rvalue; the object referred to is never moved from by these operations.

8.1.8.8 Equality operators

[expected.ref.eq]

- ¹ The equality operators behave as specified for the primary template (8.1.6.8), comparing referents through `operator*` and errors through `error()`.

9 Acknowledgments

`expected<T&, E>` is a direct extension of the work on `optional<T&>` [P2988R12] by Steve Downey and Peter Sommerlad. Many thanks to JeanHeyd Meneide for the foundational reference research [P1683R0, REFBIND], and to the contributors to the Beman Project’s `beman.expected` reference implementation [Downey_beman_expected].

Document history

— R0 Initial revision.

References

- [D4270R0] Stephen Downey. Free value or else. https://github.com/steve-downey/free_value_or_tree/5060f1e5c22db1e36db0bdb270e4e0efdcdb575d, 2026. Unpublished draft D4270R0, commit 5060f1e5c22db1e36db0bdb270e4e0efdcdb575d.
- [Downey_beman_expected] Stephen Downey. `beman.expected`. <https://github.com/bemanproject/expected>.
- [P0323R12] Vicente Botet, JF Bastien, and Jonathan Wakely. P0323R12: `std::expected`. <https://wg21.link/p0323r12>, 1 2022.
- [P1683R0] JeanHeyd Meneide. P1683R0: References for standard library vocabulary types - an optional case study. <https://wg21.link/p1683r0>, 2 2020.
- [P2255R2] Tim Song. P2255R2: A type trait to detect reference binding to temporary. <https://wg21.link/p2255r2>, 10 2021.
- [P2573R2] Yihe Li. P2573R2: `= delete`(“should have a reason”);. <https://wg21.link/p2573r2>, 3 2024.
- [P2988R12] Steve Downey and Peter Sommerlad. P2988R12: `std::optional<t&>`. <https://wg21.link/p2988r12>, 4 2025.
- [REFBIND] JeanHeyd Meneide. To bind and loose a reference | the pasture. <https://thephd.dev/to-bind-and-loose-a-reference-optional>. (Accessed on 06/26/2026).