A view of 0 or 1 elements: `views::nullable`
And a concept to constrain `maybes`

Steve Downey (sdowney@gmail.com)

Document #: P1255R14
Date: 2024-10-16
Project: Programming Language C++
Audience: LEWG

Abstract

This paper proposes a range adaptor `views::nullable` which adapts nullable types—such as `std::optional` or pointer to object types—into a range of the underlying type.

It also proposes a concept `maybe`, that types which are contextually convertible to bool and dereferenceable model.

Contents

Changes Since Last Version

— **Changes since R13**,
  — Specification of free functions

— **Changes since R12**,
  — Removal of views::maybe in favor of range optional
  — `std::maybe` concept
  — Free functions on `maybe` types such as optional and pointers.
— **Changes since R12**,
  — Paper system issues—No text changes over R11

1 Before / After Table

```
auto opt = possible_value();
if (opt) {
    // a few dozen lines ...
    use(*opt); // is *opt Safe ?
}
```

```
for (auto&& opt :
     views::nullable(possible_value())) {
    // a few dozen lines ...
    use(opt); // opt is Safe
}
```

```
std::optional o{7};
if (o) {
    *o = 9;
    std::cout << "o=" << *o << " prints 9\n";
}
std::cout << "o=" << *o << " prints 9\n";
```

```
std::optional o{7};
for (auto&& i
    : views::nullable_view<std::optional<int>&>(o)) {
    i = 9;
    std::cout << "i=" << i << " prints 9\n";
}
std::cout << "o=" << *o << " prints 9\n";

// if range for is too much magic
if (auto v = views::nullable_view<std::optional<int>&>(o);
    std::begin(v) != std::end(v)) {
    auto itr = std::begin(v);
    *itr     = 10;
    std::cout << "*itr=" << *itr << " prints 10\n";
}
std::cout << "o=" << *o << " prints 10\n";
```

```
std::vector<int> v{2, 3, 4, 5, 6, 7, 8, 9, 1};
auto test = [](int i) -> std::optional<int> {
    switch (i) {
    case 1:
    case 3:
    case 7:
    case 9:
        return i;
    default:
        return {};
    }
};

auto&& r = v | ranges::views::transform(test) |
           ranges::views::filter(
               [](auto x) { return bool(x); }) |
           ranges::views::transform(
               [](auto x) { return *x; }) |
           ranges::views::transform([](int i) {
               std::cout << i;
               return i;
           });
for (auto&& i : r) {
};
```

```
std::vector<int> v{2, 3, 4, 5, 6, 7, 8, 9, 1};
auto test = [](int i) -> std::optional<int> {
    switch (i) {
    case 1:
    case 3:
    case 7:
    case 9:
        return i;
    default:
        return {};
    }
};

auto&& r =
    v | ranges::views::transform(test) |
    ranges::views::transform(views::nullable) |
    ranges::views::join |
    ranges::views::transform([](int i) {
        std::cout << i;
        return i;
    });
for (auto&& i : r) {
};
```

```
std::vector<int> v{2, 3, 4, 5, 6, 7, 8, 9, 1};

auto test = [](int i) -> optional<int> {
    switch (i) {
    case 1:
    case 3:
    case 7:
    case 9:
        return optional{i};
    default:
        return optional<int>{};
    }
};

auto&& r = v | ranges::views::transform(test) |
           ranges::views::join |
           ranges::views::transform([](int i) {
               std::cout << i;
               return i;
           });
for (auto&& i : r) {
};
```

2 Motivation

In writing range transformation it is useful to be able to lift a value into a view that is either empty or contains the value. `std::optional` fills this function. For nullable types, types which model `maybe`, constructing an empty view for disengaged values and providing a view to the underlying value is useful as well.

```
    int fortytwo = 42;
    int sixbynine = 54;

    std::vector<int*> v{&fortytwo, nullptr, &sixbynine};

    auto r = std::views::join(std::views::transform(v, views::nullable));

    for (auto i : r) {
        std::cout << i << '\t'; // prints 42 and 54 skipping the nullptr
    }
    std::cout << std::endl;
```

The `maybe` protocol that `views::nullable` adapts is inherently unsafe because it models unsafe pointer semantics. If a nullable type is disengaged, using the derefence operator `operator*()` is undefined behavior, in practice reading from a union with the wrong type engaged. The allowed operations of `views::nullable` are all, in themselves, safe, and using the adapter can lead to safer code.

An example is using `views::nullable` in range based for loops, allowing the contained nullable value to not be dereferenced within the body. This is of small value in small examples in contrast to testing the nullable in an if statement, but with longer bodies the dereference is often far away from the test. This can be a particular issue in doing code reviews where the test, if it exists, is not visible. Often the first line in the body of the `if` is naming the dereferenced nullable, and lifting the dereference into the `for` loop eliminates some boilerplate code, the same way that range based `for` loops do.

```
{
    auto opt = possible_value();
    if (opt) {
        // a few dozen lines ...
        use(*opt); // is *opt Safe ?
    }
}

for (auto&& opt :
     views::nullable(possible_value())) {
    // a few dozen lines ...
    use(opt); // opt is Safe
}
```

Of course, if the `maybe` is empty, there is nothing in the view to act on.

```
auto oe = std::optional<int>{};
for (int i : views::nullable(oe))
    std::cout << "i=" << i
              << '\n'; // does not print
```

Converting an optional type into a view can make APIs that return optional types, such as lookup operations, easier to work with in range pipelines.

```
std::unordered_set<int> set{1, 3, 7, 9};

auto flt = [=](int i) -> std::optional<int> {
    if (set.contains(i))
        return i;
    else
        return {};
};

for (auto i : ranges::iota_view{1, 10} |
                  ranges::views::transform(flt)) {
    for (auto j : views::nullable(i)) {
        for (auto k : ranges::iota_view(0, j))
            std::cout << '\a';
        std::cout << '\n';
    }
}
```

3 Borrowed Range

A `borrowed_range` is one whose iterators cannot be invalidated by ending the lifetime of the range.

The reference specializations of `nullable_view` are borrowed. Iterators refer to the underlying object directly.

All instantiations of `nullable_view` over a pointer to object are borrowed ranges. The iterator refers to the address of the object pointed to without involving any addresss in the view.

A `nullable_view<shared_ptr>`, however, is not a borrowed range, as it participates in ownership of the `shared_ptr` and might invalidate the iterators if upon the end of its lifetime it is the last owner.

An example of code that is enabled by borrowed ranges, if unlikely code:

```
num = 42;
int k = *std::ranges::find(views::nullable(&num), num);
```

Providing the facility is not a significant cost, and conveys the semantics correctly, even if the simple examples are not hugely motivating. In particular there is no real implementation impact, other than providing template variable specializations for enable_borrowed_range.

4 Monadic Operations

It is not feasible to give `nullable_view` a direct monadic interface as it would require, for example, construction of a type that dereferences to `U` to support `transform` over `T -> U`, but that is not in general possible in the C++ type system. The additional level of indirection built in to `nullable_view` makes this infeasible.

4.1 Reference Specialization

Having worked with the `reference_wrapper` support for some time, the ergonomics are somewhat lacking. In addition, many of the Big Dumb Business Objects that are the result of lookups, or filters, and are thus good candidates for optionality, are also not good at move operations, having dozens of individual members that are a mix of primitives, strings, and sub-BDOs, resulting in complex move constructors. In addition, many old and well tested functions will mutate these objects, rather than making copies, using a more object oriented than value oriented style.

For these reasons, supporting the common case of reference semantics ergonomically is important. Folding the implementation of `reference_wrapper` into a template specialization for `T&` provides good ergonomics. The type `nullable_view` does not support assignment from the underlying type, so the only question for semantics is assignment from another instance of the same type. The semantics of `std::reference_wrapper` are well established and correct, where the implementation pointer is reassigned, putting the assignee into the same state as the assigned. These are the same semantics currently proposed for `optional<T&>`.

The range adaptor `views::nullable` only produces the non-reference specialization. As range code is strongly rooted in value semantics, providing reference semantics without ceremony seems potentially dangerous. Having it dependent on the value category of the expression would make it far too easy to produce reference types that would dangle. If the pattern becomes common, providing an instance of the function object with a distinct name would be non-breaking for anyone. Using the constructor form is not a particular burden.

This means that all of the operations on `nullable_view` are directly safe. To construct a non-safe operation is possible, but looks unsafe in code. For example:

```
nullable_view<int> o1{42};
// ...
assert(*(o1.data()) == 42));
```

Dereferencing the result of `data()` without a check for null is of course unsafe, but in a way that should be visible to programmers, reviewers, and tools.

5 Maybe Concept and Functions

The `nullable_view` adapts types that model `maybe`. In addition to the adapter, this paper propsoses exposing the concept and the most generally used helper functions for `maybes`

- `value_or`—The referred to value or an alternative
- `reference_or`—A reference to the referred to value or a reference to an alternative
- `or_invoke`—The referred to value or a function to invoke to return a value
- `yield_if`—An optional holding a value or not conditioned on a boolean

These mirror functions commonly found in optional implementations, however most of the implementations were specified before traits like `common_reference` were standardised, and can not be fixed today. For example `std::optional<T>::value_or(U)` returns a T.

The concept is very minimal, as most uses of a `maybe` impose few constraints.

```
template <class T>
concept maybe = requires(const T t) {
    bool(t);
    *(t);
};
```

Figure 1: concept maybe

This ensures that a user type provides a conversion to `bool` and a derefence operator that can be used for `const` qualified types.

The implementation of each of the functions is relatively straightforward, albiet somewhat anoying.

```
template <class T, class R = optional<decay_t<T>>>
constexpr auto yield_if(bool b, T&& t) -> R {
    return b ? forward<T>(t) : R{};
}
```

Figure 2: yield_if

6 Design

The type `nullable_view` has the semantics of zero or one objects based on if the underlying nullable object does or does not have a value, as indicated by the `maybe` being truthy or falsy.

```
template <maybe T,
          class U,
          class R = common_reference_t<iter_reference_t<T>, U&&>>
constexpr auto reference_or(T&& m, U&& u) -> R {
    static_assert(!reference_constructs_from_temporary_v<R, U>);
    static_assert(!reference_constructs_from_temporary_v<R, T&>);
    return bool(m) ? static_cast<R>(*m) : static_cast<R>((U&&)u);
}
```

Figure 3: reference_or

```
template <maybe T,
          class U,
          class R = common_type_t<iter_reference_t<T>, U&&>>
constexpr auto value_or(T&& m, U&& u) -> R {
    return bool(m) ? static_cast<R>(*m) : static_cast<R>(forward<U>(u))
}
```

Figure 4: value_or

```
template <maybe T,
          class I,
          class R = common_type_t<iter_reference_t<T>,
                                  invoke_result_t<I>>>
constexpr auto or_invoke(T&& m, I&& invocable) -> R {
    return bool(m) ? static_cast<R>(*m) : static_cast<R>(invocable());
}
```

Figure 5: or_invoke

7 Freestanding

The type `nullable_view` naturally meets the requirements for freestanding, as do the concept and proposed functions.

8 Implementation

A publicly available implementation at `https://github.com/steve-downey/view_maybe`. There are no particular implementation difficulties or tricks. The declarations are essentially what is quoted in the Wording section and the implementations are described as effects.

9 Proposal

Add a range adaptor object

`views::nullable` a range adaptor over a `nullable_object` producing a view into the nullable object.

A `maybe` object is one that is both contextually convertible to bool and for which the type produced by dereferencing is an equality preserving object. Non void pointers, `std::optional`, and the proposed `std::expected` [P0323R9] types all model *`maybe`*. Function pointers do not, as functions are not objects. Iterators do not generally model *`maybe`*, as they are not required to be contextually convertible to bool.

The generic type `std::nullable_view`, which is produced by `views::nullable` is further specialized over reference types, such that operations on the iterators of the range modify the object the range is over, if and only if the object exists.

10 Wording

Modify 22 General utilities library[utilities]

𝔅.1 Maybe Types [maybe]

𝔅.𝔅.1 In general [maybe.general]

1 Subclause 𝔅.1 describes the concept `maybe` that represents types that model optionality and functions that operate on such types.

𝔅.𝔅.2 Header `<maybe>` synopsis [maybe.syn]

```
namespace std {
  // ??, concept maybe
template <class T>
concept maybe = requires(const T t) {
    bool(t);
    *(t);
  };

template <class R = void, maybe T, class... U>
    constexpr auto value_or(T&& m, U&&... u) -> see below;

template <class R = void, maybe T, class IL, class... U>
    constexpr auto value_or(T&& m, initializer_list<IL> il, U&&... u) -> see below;

template <class R = void, maybe T, class U>
    constexpr auto reference_or(T&& m, U&& u) -> see below;

template <class R = void, maybe T, class I>
    constexpr auto or_invoke(T&& m, I&& invocable) -> see below;

}
```

```
template <class R = void, maybe T, class... U>
    constexpr auto value_or(T&& m, U&&... u) -> see below;
```

1 If `R` is not void, the return type, `RT` is `R`.

Otherwise,

if sizeof...(U) is 1,

`RT` is `common_type_t<remove_cvref_t<decltype(*std::forward<T>(m))>>`.

Otherwise, `RT` is `remove_cvref_t<decltype(*std::forward<T>(m))>`.

2 *Mandates*:

(2.1) — `RT`, is a valid type.

(2.2) — `is_constructible_v<RT, decltype(*std::forward<T>(m))> && is_constructible_v<RT, U...>` is true.

(2.3) — If `RT` is a reference type, then

(2.3.1) — `sizeof(U...)` is 1 and

(2.3.2) — `reference_constructs_from_temporary_v<RT, decltype(*std::forward<T>(m))> || reference_constructs_from_temporary_v<RT, U...>` is false.

3 *Effects*: Equivalent to:

```
return bool(m) ? static_cast<RT>(*forward<T>(m)) : RT(forward<U>(u)...);
```

```
template <class R = void, maybe T, class IL, class... U>
    constexpr auto value_or(T&& m, initializer_list<IL> il, U&&... u) -> see below;
```

4 If `R` is not void, the return type, `RT` is `R`,

otherwise `RT` is `remove_cvref_t<decltype(*std::forward<T>(m))>`.

5 *Mandates*: `is_constructible_v<RT, decltype(*std::forward<T>(m))> && is_constructible_v<RT, initializer_list<IT> il, U...>` is true.

6 *Effects*: Equivalent to:

```
return bool(m) ? static_cast<RT>(*forward<T>(m)) : RT(il, forward<U>(u)...);
```

```
template <class R = void, maybe T, class U>
    constexpr auto reference_or(T&& m, U&& u) -> see below;
```

7 If R is not void, the return type, RT is R,
otherwise RT is common_reference_t<decltype(*std::forward<T>(m)), U&&>.

8 *Mandates*:

(8.1) — RT is a valid type.

(8.2) — reference_constructs_from_temporary_v<RT, U> ||
reference_constructs_from_temporary_v<RT, decltype(*std::forward<T>(m)> is false.

9 *Effects*: Equivalent to:

```
return  bool(m) ?
  static_cast<RT>(*std::forward<T>(m)) :
  static_cast<RT>(std::forward<U>(u))
```

```
template <class R = void, maybe T, class I>
    constexpr auto or_invoke(T&& m, I&& invocable) -> see below;
```

10 If R is not void, the return type, RT is R,
otherwise RT is common_type_t<decltype(*std::forward<T>(m)), invoke_result_t<I>>.

11 *Mandates*:

(11.1) — Where DerefType is decltype(*forward<T>(m)) and

(11.2) — InvokeType is std::invoke_result_t<I>

(11.3) — common_type<DerefType, invoke_result_t<I>>::type is defined and is Ret

(11.4) — is_constructible_v<Ret, DerefType> is true

(11.5) — is_constructible_v<Ret, InvokeType> is true

(11.6) — reference_constructs_from_temporary_v<Ret, DerefType> is false

(11.7) — reference_constructs_from_temporary_v<Ret, InvokeType> is false

12 *Effects*: Equivalent to:

```
return bool(m) ?
  static_cast<RT>(*m) :
  static_cast<RT>(std::forward<I>(invocable)());
```

Modify 22.5 Optional objects[optional]

Add to 22.5.2 Header <optional> synopsis

```
template <class T, class R>
constexpr auto yield_if(bool b, T&& t) -> R;
```

Add 22.5.X

🟥.🟥.3 yield_if [optional.yield_if]

```
template <class T, class R = optional<decay_t<T>>>
constexpr auto yield_if(bool b, T&& t) -> R;
```

1 *Returns*: b ? forward<T>(t) : R{nullopt}.

Modify 26.2 Header <ranges> synopsis

🟥.2 Header <ranges> synopsis [ranges.syn]

```
// 🟥.🟥.1, nullable view
template<typename T>
requires see below;
class nullable_view;                                    // freestanding
```

```
template<class T>
constexpr bool
enable_borrowed_range<nullable_view<T*>> = true;  // freestanding

template<class T>
constexpr bool
enable_borrowed_range<nullable_view<T&>> = true;  // freestanding

namespace views {
  inline constexpr unspecified nullable = unspecified;          // freestanding
}
```

𝔉.𝔉.1 Nullable View [range.nullable]

𝔉.𝔉.1.1 Overview [range.nullable.overview]

1 `nullable_view` is a range adaptor that produces a `view` with cardinality 0 or 1. It adapts object types which model the exposition only concept `nullable_object`.

2 The name `views::nullable` denotes a customization point object (**??**). Given a subexpression `E`, the expression `views::nullable(E)` is expression-equivalent to `nullable_view<decay_t<decltype((E))>>(E)`.

𝔉.𝔉.1.2 Class template `nullable_view` [range.nullable.view]

[*Example 1*:

```
optional o{4};
for (int k : nullable_view m{o})
  cout << k;        // prints 4
```

— *end example*]

𝔉.𝔉.1.3 Class template `nullable_view` [range.nullable.view]

```
namespace std::ranges {
  template<class I>
  concept readable-references = see below;  // exposition only

  template<class I>
    concept nullable-object = see below;  // exposition only

  template <class T>
    concept movable-object = see below;  // exposition only

  template <typename Nullable>
    requires(movable-object<Nullable>
  class nullable_view<Nullable>
    : public ranges::view_interface<nullable_view<Nullable>> {
  private:
    using T = remove_reference_t<
        iter_reference_t<typename unwrap_reference_t<Nullable>>>;

    movable-box<Nullable> value;  // exposition only (see ??)

  public:
    constexpr nullable_view() = default;

    constexpr explicit nullable_view(Nullable const& nullable);

    constexpr explicit nullable_view(Nullable&& nullable);
```

```
    template <class... Args>
        requires constructible_from<Nullable, Args...>
    constexpr nullable_view(in_place_t, Args&&... args);

    constexpr T*       begin() noexcept;
    constexpr const T* begin() const noexcept;
    constexpr T*       end() noexcept;
    constexpr const T* end() const noexcept;

    constexpr size_t size() const noexcept;

    constexpr auto data() noexcept;

    constexpr const auto data() const noexcept;

    friend constexpr auto operator<=>(const nullable_view& l,
                                      const nullable_view& r);

    friend constexpr bool operator==(const nullable_view& l,
                                     const nullable_view& r);
};
```

1 The exposition-only *readable-references* concept is equivalent to:

```
template<class Ref, class ConstRef>
concept readable-references =                // exposition only
is_lvalue_reference_v<Ref> &&
is_object_v<remove_reference_t<Ref>> &&
is_lvalue_reference_v<ConstRef> &&
is_object_v<remove_reference_t<ConstRef>> &&
convertible_to<add_pointer_t<ConstRef>, const remove_reference_t<Ref>*>;
```

2 The exposition-only *nullable-object* concept is equivalent to:

```
template<class I>
concept nullable-object =                // exposition only
is_object_v<I> &&
maybe<T> &&
readable-references<std::iter_reference_t<T>,
                    std::iter_reference_t<const T>>;
```

3 When an object is contextually convertible to `bool` and dereferencable via `operator *()`, the object is said to be a *nullable object*.

```
constexpr explicit nullable_view();
```

4 *Effects*: Initializes *value* with `nullptr`

```
constexpr explicit nullable_view(Nullable nullable);
```

5 *Effects*: Initializes *value* with `addressof(nullable)`

```
constexpr T* begin() noexcept;
constexpr const T* begin() const noexcept;
```

6 *Returns*: `data();`

```
constexpr T* end() noexcept;
constexpr const T* end() const noexcept;
```

7 *Returns*: `data() + size();`.

```
static constexpr size_t size() noexcept;
```

8 *Effects*: Equivalent to:

```
if (!value)
  return 0;
Nullable& m = *value;
```

```
      return bool(m);
```

```
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;
```

10 *Effects*: Equivalent to:

```
      if (!value)
        return nullptr;
      const Nullable& m = *value;
      return m ? addressof(*m) : nullptr;
```

𝔷.𝔷.1.4 Class template specialization `nullable_view<T&>` [range.nullable.view.ref]

```
template <typename Nullable>
  requires(nullable-object<Nullable>
class nullable_view<Nullable&>
    : public ranges::view_interface<nullable_view<Nullable>> {
  private:
    using T = remove_reference_t<
        iter_reference_t<typename unwrap_reference_t<Nullable>>>;

    Value*  value ; // exposition only

  public:
    constexpr nullable_view() : value(nullptr){};

    constexpr explicit nullable_view(Nullable& nullable);

    constexpr explicit nullable_view(Nullable&& nullable) = delete;

    constexpr T*       begin() noexcept;
    constexpr const T* begin() const noexcept;
    constexpr T*       end() noexcept;
    constexpr const T* end() const noexcept;

    constexpr size_t size() const noexcept;

    constexpr T* data() noexcept;

    constexpr const T* data() const noexcept;
};
```

1 `constexpr explicit nullable_view();`

2 *Effects*: Initializes *value* with `nullptr`

`constexpr explicit nullable_view(Nullable nullable);`

3 *Effects*: Initializes *value* with `addressof(nullable)`

```
constexpr explicit nullable_view(Nullable&& nullable) = delete;
```

4 Deleted

```
constexpr T* begin() noexcept;
constexpr const T* begin() const noexcept;
```

5 *Returns*: `data();`

```
constexpr T* end() noexcept;
constexpr const T* end() const noexcept;
```

6 *Returns*: `data() + size();`.

`static constexpr size_t size() noexcept;`

7 *Effects*: Equivalent to:

```
    if (!value)
      return 0;
    return bool(*value);
```

```
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;
```

8 *Effects*: Equivalent to:

```
    if (!value)
      return nullptr;
    return *value ? addressof(**value) : nullptr;
```

𝔉.𝔉.2 Feature-test macro [version.syn]

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

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

11 Impact on the standard

A pure library extension, affecting no other parts of the library or language.
The proposed changes are relative to the current working draft [N4958].

Document history

- **R12**,
  - Paper system title issues - no text changes
- **Changes since R11**,
  - Expand on design and implementation details
  - Monadic functions use deducing `this`
  - Constraints, Mandates, Returns, Effects clean up
- **Changes since R10**,
  - Complete History in history section
  - exposid formatting and ampersand escaping TeX formatting nits
- **Changes since R9**,
  - Fix Borrowed Ranges post naming split
  - Clarify safety concerns
- **Changes since R8**,
  - Give maybe and nullable distinct template names
  - Propose T& specializations
  - Propose monadic interface for maybe_view
  - Wording++
  - Freestanding
- **Changes since D7**, presented to SG9 on 2022.07.11
  - Layout issues
  - References include paper source
  - Citation abbreviation form to 'abstract'
  - 'nuulable' typo fix
  - Markdown backticks to tcode
  - ToC depth and chapter numbers for Ranges
  - No technical changes to paper — all presentation
- **Changes since R7**
  - Update all Wording.
  - Convert to standards latex macros for wording.
  - Removed discussion of list comprehension desugaring - will move to yield_if paper.
- **Changes since R6**
  - Extend to all object types in order to support list comprehension
  - Track working draft changes for Ranges
  - Add discussion of _borrowed_range_
  - Add an example where pipelines use references.
  - Add support for proxy references (explore std::pointer_traits, etc).
  - Make std::views::maybe model std::ranges::borrowed_range if it's not holding the object by value.

    - Add a const propagation section discussing options, existing precedent and proposing the option that the author suggests.
- **Changes since R5**
    - Fix reversed before/after table entry
    - Update to match C++20 style [@N4849] and changes in Ranges since [@P0896R3]
    - size is now size_t, like other ranges are also
    - add synopsis for adding to ‘<ranges>‘ header
    - Wording clean up, formatting, typesetting
    - Add implementation notes and references
- **Changes since R4**
    - Use std::unwrap_reference
    - Remove conditional ‘noexcept‘ness
    - Adopted the great concept renaming
- **Changes since R3**
    - Always Capture
    - Support reference_wrapper
- **Changes since R2**
    - Reflects current code as reviewed
    - Nullable concept specification
    - Remove Readable as part of the specification, use the useful requirements from Readable
    - Wording for views::maybe as proposed
    - Appendix A: wording for a view_maybe that always captures
- **Changes since R1**
    - Refer to views::all
    - Use wording ’range adaptor object’
- **Changes since R0**
    - Remove customization point objects
    - Concept ‘Nullable‘, for exposition
    - Capture rvalues by decay copy
    - Remove maybe_view as a specified type

References

[N4958] Thomas Köppe. N4958: Working draft, programming languages — c++. `https://wg21.link/n4958`, 8 2023.

[P0323R9] JF Bastien and Vicente Botet. P0323R9: std::expected. `https://wg21.link/p0323r9`, 8 2019.

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

[viewmayb27:online] Steve Downey. A view of 0 or 1 elements: views::maybe. `https://github.com/steve-downey/view_maybe/blob/master/papers/view-maybe.tex`, 07 2022. (Accessed on 08/15/2022).

12 Possible Wording

Modify 26.2 Header <ranges> synopsis

🔲.1 Header `<ranges>` synopsis [ranges.syn]

🔲.🔲.1 Possible View [range.possible]

🔲.🔲.1.1 Overview [range.possible.overview]

1 `possible_view` produces a view that contains 0 or 1 objects.

2 The name `views::possible` denotes a customization point object (**??**). Given a subexpression `E`, the expression `views::possible(E)` is expression-equivalent to `possible_view<decay_t<decltype((E))>>(E)`.

[*Example 1*:

```
int i{4};
for (int i : views::possible(4))
  cout << i;        // prints 4

possible_view<int> m2{};
for (int k : m2)
  cout << k;        // Does not execute
```

— *end example*]

🔲.🔲.1.2 Class template `possible_view` [range.possible.view]

```
template <typename Value>
class possible_view;


template <typename Value>
class possible_view : public ranges::view_interface<possible_view<Value>> {
  private:
    std::optional<Value> value_;              // exposition only

  public:
    constexpr possible_view() = default;

    constexpr explicit possible_view(const Value& value) requires copy_constructible<T>;

    constexpr explicit possible_view(Value&& value);

    template <class... Args>
      requires constructible_from<T, Args...>
    constexpr possible_view(std::in_place_t, Args&&... args);

    constexpr Value*       begin() noexcept;
    constexpr const Value* begin() const noexcept;
    constexpr Value*       end() noexcept;
    constexpr const Value* end() const noexcept;

    constexpr size_t size() const noexcept;

    constexpr Value* data() noexcept;

    constexpr const Value* data() const noexcept;

    friend constexpr auto operator<=>(const possible_view& lhs,
    const possible_view& rhs) {
      return lhs.value_ <=> rhs.value_;
    }

    friend constexpr bool operator==(const possible_view& lhs,
    const possible_view& rhs) {
      return lhs.value_ == rhs.value_;
    }
```

```
    template <typename Self, typename F>
    constexpr auto and_then(this Self&& self, F&& f);

    template <typename Self, typename F>
    constexpr auto transform(this Self&& self, F&& f);

    template <typename Self, typename F>
    constexpr auto or_else(this Self&& self, F&& f);
};

constexpr explicit possible_view(Value const& possible);
```

1 *Effects*: Initializes *value_* with possible.

```
constexpr explicit possible_view(Value&& possible);
```

2 *Effects*: Initializes *value_* with `std::move(possible)`.

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

3 *Effects*: Initializes *value_* as if by *value_*`{in_place, std::forward<Args>(args)...}`.

```
constexpr T* begin() noexcept;
constexpr const T* begin() const noexcept;
```

4 *Returns*: `data()`;

```
constexpr T* end() noexcept;
constexpr const T* end() const noexcept;
```

5 *Returns*: `data() + size();`.

```
static constexpr size_t size() noexcept;
```

6 *Returns*:

```
  bool(value_);
```

```
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;
```

7 *Returns*: `addressof(*value_);`

```
constexpr auto operator<=>(const possible_view& lhs, const possible_view& rhs)
```

8 *Returns*: `lhs.value_ <=> rhs.value_;`

```
constexpr auto operator==(const possible_view& lhs, const possible_view& rhs)
```

9 *Returns*: `lhs.value_ == rhs.value_;`

```
template <typename Self, typename F>
constexpr auto and_then(this Self&& self, F&& f);
```

10 Let `U` be `invoke_result_t<F, decltype(forward_like<Self>(*self.value_))>`.

11 *Mandates*: `remove_cvref_t<U>` is a specialization of `possible_view`.

12 *Effects*: Equivalent to:

```
if (self.value_) {
  return std::invoke(std::forward<F>(f),
                     forward_like<Self>(*self.value_));
} else {
  return std::remove_cvref_t<U>();
}
```

```
template <typename Self, typename F>
constexpr auto transform(this Self&& self, F&& f);
```

13 Let `U` be `invoke_result_t<F, decltype(forward_like<Self>(*self.value_))>`.

14 *Returns*:

```
        (self.value_)
        ? possible_view<U>{std::invoke(std::forward<F>(f),
                                       forward_like<Self>(*self.value_))}
        : possible_view<U>{};

template <typename Self, typename F>
constexpr auto or_else(this Self&& self, F&& f);
```

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

Mandates: `remove_cvref_t<U>` is a specialization of `possible_view`.

Returns:

```
self.value_ ? std::forward<Self>(self) : std::forward<F>(f)();
```

𝔉.𝔉.1.3 Class template specialization`possible_view<T&>` [range.possible.view.ref]

```
template <typename Value>
class possible_view<Value&> : public ranges::view_interface<possible_view<Value&>> {
  private:
    Value*  value_ ; // exposition only

  public:
    constexpr possible_view();

    constexpr explicit possible_view(Value& value);

    constexpr explicit possible_view(Value&& value) = delete;

    constexpr Value*       begin() noexcept;
    constexpr const Value* begin() const noexcept;
    constexpr Value*       end() noexcept;
    constexpr const Value* end() const noexcept;

    constexpr size_t size() const noexcept;

    constexpr Value* data() noexcept;

    constexpr const Value* data() const noexcept;

    friend constexpr auto operator<=>(const possible_view& lhs,
                                      const possible_view& rhs);

    friend constexpr bool operator==(const possible_view& lhs,
                                     const possible_view& rhs);

    template <typename Self, typename F>
    constexpr auto and_then(this Self&& self, F&& f);

    template <typename Self, typename F>
    constexpr auto transform(this Self&& self, F&& f);

    template <typename Self, typename F>
    constexpr auto or_else(this Self&& self, F&& f);
};
```

```
constexpr explicit possible_view();
```

Effects: Initializes *value_* with `nullptr`

```
constexpr explicit possible_view(Value possible);
```

Effects: Initializes *value_* with `addressof(possible)`

```
constexpr T* begin() noexcept;
constexpr const T* begin() const noexcept;
```

Returns: `data()`;

```
constexpr T* end() noexcept;
constexpr const T* end() const noexcept;
```

4 *Returns*: `data() + size();`.

```
static constexpr size_t size() noexcept;
```

5 *Returns*:

```
bool(value_);
```

```
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;
```

6 *Effects*: Equivalent to:

```
if (!value_)
    return nullptr;
return addressof(*value_);
```

```
friend constexpr auto operator<=>(const possible_view& lhs,
                                  const possible_view& rhs);
```

7 *Returns*:

```
(bool(lhs.value_) && bool(rhs.value_))
    ? (*lhs.value_ <=> *rhs.value_)
    : (bool(lhs.value_) <=> bool(rhs.value_));
```

```
friend constexpr auto operator==(const possible_view& lhs,
                                 const possible_view& rhs);
```

8 *Returns*:

```
(bool(lhs.value_) && bool(value_))
    ? (*lhs.value_ == *rhs.value_)
    : (bool(lhs.value_) == bool(rhs.value_));
```

```
template <typename Self, typename F>
constexpr auto and_then(this Self&& self, F&& f);
```

9 Let `U` be `invoke_result_t<F, decltype(forward_like<Self>(*self.value_))>`.

10 *Mandates*: `remove_cvref_t<U>` is a specialization of `possible_view<T&>`.

11 *Effects*: Equivalent to:

```
if (self.value_) {
  return std::invoke(forward<F>(f),
                     forward_like<Self>(*self.value_));
} else {
  return std::remove_cvref_t<U>();
}
```

```
template <typename Self, typename F>
constexpr auto transform(this Self&& self, F&& f);
```

12 Let `U` be `invoke_result_t<F, decltype(forward_like<Self>(*self.value_))>`.

13 *Returns*:

```
return (self.value_)
   ? possible_view<U>{std::invoke(forward<F>(f),
                                  forward_like<Self>(*self.value_))}
   : possible_view<U>{};
```

```
template <typename Self, typename F>
constexpr auto or_else(this Self&& self, F&& f);
```

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

14 *Mandates*: `remove_cvref_t<U>` is a specialization of `possible_view<T&>`.

15 *Returns*:

```
return self.value_ ? std::forward<Self>(self) : std::forward<F>(f)();
```