A view of 0 or 1 elements: `views::maybe`

Steve Downey (sdowney@gmail.com)

Document #: P1255R8
Date: 2022-07-11
Project: Programming Language C++
Audience: SG9, LEWG

Abstract

This paper proposes `views::maybe` a range adaptor that produces a view with cardinality 0 or 1 which adapts copyable object types, values, and nullable types such as `std::optional` and pointer to object types.

Contents

1 Before / After Table

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

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

```
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::maybe(std::ref(o))) {
 i = 9;
  std::cout << "i=" << i << " prints 9\n";
}
std::cout << "o=" << *o << " prints 9\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;
    });
```

```
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::maybe) |
    ranges::views::join |
    ranges::views::transform([](int i) {
        std::cout << i;
        return i;
    });
```

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. For types that are `nullable`, constructing an empty view for disengaged values and providing a view to the underlying value is useful as well. The adapter `views::single` fills a similar purpose for non-nullable values, lifting a single value into a view, and `views::empty` provides a range of no values of a given type. The type `views::maybe` can be used to unify `single` and `empty` into a single type for further processing. This is in particuluar useful when translating list comprehensions.

```
std::vector<std::optional<int>> v{
    std::optional<int>{42},
    std::optional<int>{},
    std::optional<int>{6 * 9}};

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

for (auto i : r) {
    std::cout << i; // prints 42 and 54
}
```

In addition to range transformation pipelines, `views::maybe` can be used in range based for loops, allowing the 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. 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 OK ?
    }
}

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

The view can be on a `std::reference_wrapper`, allowing the underlying nullable to be modified:

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

Of course, if the nullable is empty, there is nothing in the view to modify.

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

Converting an optional type into a view can make APIs that return optional types, such a 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::maybe(i)) {
        for (auto k :
             ranges::iota_view(0, j))
            std::cout << '\a';
        std::cout << '\n';
    }
}
```

3 Lazy monadic pythagorean triples

Eric Niebler's pythagorean triple example, using current C++ and proposed views::maybe.

```
// "and_then" creates a new view by applying
// a transformation to each element in an
// input range, and flattening the resulting
// range of ranges. A.k.a. bind (This uses
// one syntax for constrained lambdas in
// C++20.)
inline constexpr auto and_then = [](auto&& r, auto fun) {
    return decltype(r)(r) |
           std::ranges::views::transform(std::move(fun)) |
           std::ranges::views::join;
};

// "yield_if" takes a bool and a value and
// returns a view of zero or one elements.
inline constexpr auto yield_if = [](bool b, auto x) {
    return b ? maybe_view{std::move(x)}
             : maybe_view<decltype(x)>{};
};

void print_triples() {
    using std::ranges::views::iota;
    auto triples = and_then(iota(1), [](int z) {
        return and_then(iota(1, z + 1), [=](int x) {
            return and_then(iota(x, z + 1), [=](int y) {
                return yield_if(x * x + y * y == z * z,
                                std::make_tuple(x, y, z));
            });
        });
    });

    // Display the first 10 triples
    for (auto triple :
         triples | std::ranges::views::take(10)) {
        std::cout << '(' << std::get<0>(triple) << ','
                  << std::get<1>(triple) << ','
                  << std::get<2>(triple) << ')' << '\n';
    }
}
```

The implementation of `yield_if` is essentially the type unification of `single` and `empty` into `maybe`, returning an empty on false, and a range containing one value on true.

4 Proposal

Add a range adaptor object `views::maybe`, returning a view over an object, capturing by value. For ***nullable*** objects, provide a zero size range for objects which are disengaged. A ***nullable*** 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 models ***nullable***. Function pointers do not, as functions are not objects. Iterators do not generally model ***nullable***, as they are not required to be contextually convertible to bool.

5 Borrowed Range

A `borrowed_range` is one whose iterators cannot be invalidated by ending the lifetime of the range. For `views::maybe`, the iterators are `T*`, where `T` is essentially the type of the dereferenced nullable. For raw

pointers and reference_wrapper over `nullable` types, the iterator for `maybe_view` points directly to the underlying object, and thus matches the semantics of `borrowed_range`. This means that `maybe_view` is conditionally borrowed. A `maybe_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::maybe(&num), num);
```

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

6 Design

The basis of the design is to hybridize `views::single` and `views::empty`. If the view is over a value that is not *`nullable`* it is like a single view if constructed with a value, or is of size zero otherwise. For *`nullable`* types, if the underlying object claims to hold a value, as determined by checking if the object when converted to bool is true, `begin` and `end` of the view are equivalent to the address of the held value within the underlying object and one past the underlying object. If the underlying object does not have a value, `begin` and `end` return `nullptr`. `views::maybe` also has support for `std::reference_wrapper`, allowing writes through the iterator to pass through to the object held in the wrapper.

7 Implementation

A publically available implementation at `https://github.com/steve-downey/view_maybe` based on the Ranges implementation in libstdc++. 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.

Compiler Explorer Link to Before/After Examples

8 Wording

26.2 Synopsis [range.synopsis]

Modify 26.2 Header <ranges> synopsis

```
// 26.2.1, maybe view
template<copy_constructible T>
requires see below;
class maybe_view;

template <typename T>
constexpr inline bool enable_borrowed_range<maybe_view<T*>> = true;

template <typename T>
constexpr inline bool enable_borrowed_range<maybe_view<reference_wrapper<T>>> = true;

namespace views { inline constexpr unspecified maybe = unspecified; }
```

26.2.1 Maybe View [range.maybe]

26.2.1.1 Overview [range.maybe.overview]

1 `maybe_view` is a range adaptor that produces a `view` with cardinality 0 or 1. It adapts `copyable` object types and *`nullable`* types. If the type is *`nullable`*, the view is empty if the *`nullable`* is empty.

2 The name `views::maybe` denotes a customization point object ([customization.point.object]). For some subexpression `E`, the expression `views::maybe<E>` is expression-equivalent to:

(2.1) — `maybe_view(E)`, the `view` specified below, if the expression is well formed, where *`decay-copy`*`(E)` is moved into the `maybe_view`

(2.2) — otherwise `views::maybe(E)` is ill-formed.

[*Note 1*: Whenever `views::maybe(E)` is a valid expression, it is a prvalue whose type models `view`. — *end note*]

3 [*Example 1*:

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

— *end example*]

26.2.1.2 Concept nullable [range.maybe.nullable]

1 Types that:

(1.1) — - are contextually convertible to `bool`

(1.2) — are dereferenceable

(1.3) — have const references which are dereferenceable

(1.4) — the `iter_reference_t` of the type and the `iter_reference_t` of the const type, will :

(1.4.1) — satisfy `is_lvalue_reference`

(1.4.2) — satisfy `is_object` when the reference is removed

(1.4.3) — for const pointers to the referred to types, satisfy `convertible_to`

(1.5) — or are a reference_wrapper around a type that satifies `nullable`

model the exposition only `nullable` concept

2 Given a value `i` of type `I`, `I` models *`nullable`* only if the expression `*i` is equality-preserving.

[*Note 1*: The expression `*i` is required to be valid via the exposition-only *`nullable`* concept. — *end note*]

3 For convience, the exposition-only concepts *`is-reference-wrapper-v`*. *`nullable`*, *`nullable_ref`*, and *`copyable_object`* are used below.

```
// exposition only

template <class T>
concept nullable =
    std::is_object_v<T> && requires(T& t, const T& ct) {
                               bool(ct);
                               *(t);
                               *(ct);
                           };

template <class T>
concept nullable_val =
    nullable<T> &&
    readable_references<std::iter_reference_t<T>,
                        std::iter_reference_t<const T>>;

template <typename, template <typename...> class>
inline constexpr bool is_v = false;

template <typename... Ts, template <typename...> class C>
inline constexpr bool is_v<C<Ts...>, C> = true;

template <class T>
concept nullable_ref = is_v<T, std::reference_wrapper> &&
                       nullable_val<typename T::type>;

template <class T>
inline constexpr bool is_reference_wrapper_v =
    is_v<T, std::reference_wrapper>;
```

```
template <class T>
concept copyable_object = (std::copy_constructible<T> &&
                           std::is_object_v<T>);
```

26.2.1.3 Class template `maybe_view` [range.maybe.view]

```
template <typename Value>
requires(copyable_object<Value>) class maybe_view
    : public ranges::view_interface<maybe_view<Value>> {
  private:
    std::optional<Value> value_; // exposition only

  public:
    constexpr maybe_view() = default;

    constexpr explicit maybe_view(Value const& value);

    constexpr explicit maybe_view(Value&& value);

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

    constexpr Value*       begin() noexcept;
    constexpr const Value* begin() const;
    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;
};
```

```
constexpr explicit maybe_view(Value const& maybe);
```

1 *Effects*: Initializes value_ with maybe.

```
constexpr explicit maybe_view(Value&& maybe);
```

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

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

3 *Effects*: Initializes value_ as if by `value_in_place, 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 *Effects*: Equivalent to:

```
return bool(value_);
```

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

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

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

8 *Effects*: Equivalent to:

```
return std::addressof(*value_);
```

```
template <typename Maybe>
requires(copyable_object<Maybe> &&
         (nullable_val<Maybe> || nullable_ref<Maybe>))
    class maybe_view<Maybe> : public ranges::view_interface<maybe_view<Maybe>> {
      private:
        using T = see below;

        copyable-box<Maybe> value_; // exposition only

      public:
        constexpr maybe_view() = default;

        constexpr explicit maybe_view(Maybe const& maybe);

        constexpr explicit maybe_view(Maybe&& maybe);

        template <class... Args>
        requires std::constructible_from<Maybe, Args...> constexpr
        maybe_view(std::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 T* data() noexcept;

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

    // For Exposition
    using T = std::remove_reference_t<
        iter_reference_t<typename unwrap_reference_t<Maybe>>>;
```

```
constexpr explicit maybe_view(Maybe const& maybe);
```

9 *Effects*: Initializes value_ with maybe.

```
constexpr explicit maybe_view(Maybe&& maybe);
```

10 *Effects*: Initializes value_ with `std::move(maybe)`.

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

11 *Effects*: Initializes value_ as if by `value_in_place, forward<Args>(args)...`.

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

12 *Returns*: `data();`.

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

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

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

14 *Effects*: Equivalent to:

```
if constexpr (*is-reference-wrapper-v*<Maybe>) {
  return bool(value_.get().get());
} else {
  return bool(value_.get());
}
```

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

15 *Effects*: Equivalent to:

```
Maybe& m = *value_;
if constexpr (*is-reference-wrapper-v*<Maybe>) {
  return m.get() ? addressof(*(m.get())) : nullptr;
} else {
  return m ? addressof(*m) : nullptr;
}
```

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

16 *Effects*: Equivalent to:

```
const Maybe& m = *value_;
if constexpr (*is-reference-wrapper-v*<Maybe>) {
  return m.get() ? addressof(*(m.get())) : nullptr;
} else {
  return m ? addressof(*m) : nullptr;
}
```

9 Impact on the standard

A pure library extension, affecting no other parts of the library or language.

The proposed changes are relative to the current working draft [N4910].

Document history

- **Changes since 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

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

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

[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 07/11/2022).