Abstract

This paper proposes two range adaptors which produce a view with cardinality 0 or 1:

— views::maybe—a range adaptor that produces an owning view holding 0 or 1 elements of an object
— views::nullable which adapts nullable types—such as std::optional or pointer to object types—into a range of the underlying type.
1 Before / After Table

Before:
```c++
auto opt = possible_value();
if (opt) {
    // a few dozen lines ...
    use(*opt); // is *opt Safe ?
}
std::optional o{7};
if (o) {
    *o = 9;
    std::cout << "o= " << *o << " prints 9\n";
}
```

After:
```c++
std::optional o{7};
for (auto&& opt : views::nullable(possible_value())) {
    // a few dozen lines ...
    use(opt); // opt is Safe
}
```

```c++
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 x < 9; }) |
    ranges::views::transform([](int i) {
        std::cout << i;
        return i;
    }) |
    ranges::views::join |
    ranges::views::transform([](int i) {
        std::cout << 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. For types that model nullable_object, 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 particular, useful when translating list comprehensions.

```c++
std::vector<int> v{2, 3, 4, 5, 6, 7, 8, 9, 1};
auto test = [](int i) -> maybe_view<int> {
    switch (i) {
    case 1:
    case 3:
    case 7:
    case 9:
        return maybe_view<i>();
    default:
        return maybe_view<int>();
    }
};
auto&& r = v | ranges::views::transform(test) |
    ranges::views::join |
    ranges::views::transform([](int i) {
        std::cout << i;
    });
for (auto&& i : r) {
}
```
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::nullable));

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

The nullable protocol that views::nullable adapts is inherently unsafe because it models unsafe
pointer semantics. If a nullable type is disengaged, using the dereference operator operator*() is
is undefined behavior. 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 a 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
}
```

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

```
std::optional o{7};
for (auto&& i : views::nullable(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 act on.

```
auto oe = std::optional<int>{};
for (int i : views::nullable(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 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 : views::iota_view{1, 10} |
     views::transform(flt)) {
    for (auto j : views::nullable(i)) {
        for (auto k : views::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. monadic bind

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. I plan to propose this function for standardization in a following paper.

This code is essentially a mechanical translation of a list, or monadic, comprehension from Python or Haskell. In Haskell there is a pure desugaring of comprehension to `bind` and `yield_if` for comprehension guard clauses. This is an open research area for C++, and again, not part of this proposal.

4 Borrowed Range

A `borrowed_range` is one whose iterators cannot be invalidated by ending the lifetime of the range. The reference specializations of both `nullable_view` and `maybe_view` are borrowed. Iterators refer to the underlying object directly.

No other `maybe_view` is necessarily a borrowed range, and is not tagged as such.

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 addresses 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:

```cpp
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`.

5 Wait, There’s More

5.1 The Argument for a Vocabulary Type

The discussion around `std::static_vector` [P0843R4] has solidified for me that `std::optional` is not a container, and making it one would be a mistake. There are too many operations that are problematic for a component that holds at most a fixed number of elements, and existing generic code will not understand that operations like `push_back` have commonly occurring failure modes.

Ranges are not containers. The operations for a fixed sized range are the same as for any range, as it is not expected that ranges mutate that way. In practice `maybe_view` is a useful interface type in addition to `std::optional`.

Just as there are use cases for having particular containers being returned from a function, there are use cases for returning near primitive ranges as explicit types. Explicit types are much easier for compilers to generate good code, and to further optimize that code. The more limited interface of `maybe_view` seems to, in practice, make intention clearer to the compiler, over `std::optional`, over adaptation or projection.

Saying what you mean directly is better.

A `std::optional` type says I will be checking if the value is engaged or disengaged, and possibly taking alternative action based on that, and that I might use a `std::optional<T>` in contexts that I would use a `T`, or that it might be used as a default parameter. The long list suggests that `std::optional<T>` is filling too many roles. A `maybe_view`, or a `nullable_view`, says that independent check will not be made, the value will have operations applied if present, and ignored otherwise. A concrete range type sets tighter expectations.

Value types, which range types are, should, if they can, provide spaceship and equivalence operators. This is straightforward to specify, and to implement for both `maybe_view` or `nullable_view.`
5.2 The Argument for Monadic Operations

The generic templated type `maybe_view` is a monad in the category of C++ types in exactly the same way that `std::optional` and `std::expected` are. [P0798R8] The operations stay strictly within the generic template type. Since it is a type that can be reasonably used on its own, it should, on its own, support all reasonable uses of the type. This should include the function application patterns of functor and monad, and in exactly the same way they have been applied to `std::optional`. The member functions are much more strongly typed than the monad in the range category, staying within the template type.

However, it is not feasible to give `nullable_view` the same 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 into `nullable_view` makes this infeasible. The inconsistency of implementability supports the direction from SG9 to separate the templates by name, rather than just by concept.

Most range types should be treated much like lambda expressions, with unnameable types. Even where it is possible to work out the type, that type may not be stable in the face of concept specialization based on the rest of the types involved. However, `maybe_view` and `nullable_view` are primitive ranges, built out of non-range types. It is natural to write functions, including lambdas, that return them, and staying within the type system can improve correctness and diagnostics when the code strays. Providing the monadic interface for the base and for the reference specialization of `nullable_view` is entirely straight-forward.

Looking at the test code for the reference implementation, we can see that usage for the non-reference specialization is very similar:

```cpp
maybe_view<int> mv{40};
auto r = mv.and_then([](int i) { return maybe_view{i + 2}; });

ASSERT_TRUE(!r.empty());
ASSERT_TRUE(r.size() == 1);
ASSERT_TRUE(r.data() != nullptr);
ASSERT_TRUE(*(r.data()) == 42);

auto r2 = mv.and_then([](int) { return maybe_view<int>{}; });
ASSERT_TRUE(r2.empty());
ASSERT_TRUE(r2.size() == 0);
ASSERT_TRUE(r2.data() == nullptr);

maybe_view<int&> mv{forty};
auto r9 = mv.transform([](int& i) { int k = i; i = 56; return k * 2; });

for (auto r: r9) {
    ASSERT_EQ(r, 80);
}

for (auto v: mv) {
    ASSERT_EQ(v, 56);
}

ASSERT_EQ(forty, 56);
```

A test stanza for the T& case suggests more interesting applications, as transform will be applied to the underlying referred to value.

```cpp
int forty{40};
maybe_view<int&> mv{forty};
auto r9 = mv.transform([](int& i) {
    int k = i;
    i = 56;
    return k * 2;
});

for (auto r: r9) {
    ASSERT_EQ(r, 80);
}

for (auto v: mv) {
    ASSERT_EQ(v, 56);
}

ASSERT_EQ(forty, 56);
```

Introducing a new optionalish type may provide a way out of the `std::optional<T&>` quagmire. There is no risk of broken code or changes in SFINAE in template instantiation. The type `maybe_view` is not useful as
a default parameter or a substitute for its underlying type. It behaves the same way that std::reference_wrapper has for more than a decade. Maybe as a name is common in the area for these sorts of types, it is not innovative.

In order to affect the underlying referenced type, not only do we need to use maybe_view<int&> explicitly, the function passed to transform must take the underlying type by reference. A significant amount of ceremony is required. Since the general direction has been and continues to be in favor of value oriented programming, making mutation require a context in a lonely place is appropriate.

5.3 The Argument for 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. Neither maybe_view or nullable_view 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. The same semantics are adopted for maybe_view or nullable_view.

The range adaptors, views::maybe and views::nullable, only produce the non-reference specialization. As range code is strongly rooted in value semantics, providing reference semantics without ceremony seems potentially dangerous. If the pattern becomes common, providing an instance of the function object with a distinct name would be non-breaking for anyone.

The owning view maybe_view is not a container, and does not try to support the full container interface. As a range with a fixed upper size, emplace and push back operations are problematic. Not supplying them is not problematic.

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

```cpp
maybe_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 both programmers and tools.

6 Design

For maybe_view, the design is a hybrid of empty_view and single_view, with the straightforward extension for reference type, holding a pointer to an existing object. For nullable_view we have the same semantics of zero or one objects, only based on if the underlying nullable object does or does not have a value.

7 Freestanding

Both maybe_view and nullable_view naturally meet the requirements for freestanding. The expository use of optional does not interfere with the ability of maybe_view to meet the requirements of freestanding.

8 Implementation

A publicly available implementation at [https://github.com/steve-downey/view_maybe](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.
The implementation, for exposition purposes, uses `std::optional` to hold the value for `maybe_view`. Implementations, to reduce the overhead of debugging implementations should probably hoist the storage and flag in a typical optional into `maybe_view`, in which case the flag should be checked first on reads with acquire/release atomic semantics, and last on writes, so as to provide a synchronization points. Although this note may be in the close neighborhood of teaching my grandmother to suck eggs.

9 Proposal

Add two range adaptor objects

- `views::maybe` a range adaptor that produces an owning view holding 0 or 1 elements of an object.
- `views::nullable` a range adaptor over a `nullable_object` producing a view into the nullable object.

A `nullable_object` 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` types all model `nullable_object`. 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.

The generic types `std::maybe_view` and `std::nullable_view`, which are produced by `views::maybe` and `views::nullable`, respectively, are 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 26.2 Header `<ranges>` synopsis

### Header `<ranges>` synopsis

```
// 9.1 maybe view
template<move_constructible T>
requires see below;
class maybe_view;

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

template<class T>
constexpr bool enable_borrowed_range<maybe_view<reference_wrapper<T>>> = true; // freestanding

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

namespace views {
    inline constexpr unspecified maybe = unspecified; // freestanding
}

// 9.2 nullable view
template<move_constructible 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<reference_wrapper<T>>> = true; // freestanding

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

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

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

### 1.1 Maybe View

1. **Overview**

   maybe_view produces a view that contains 0 or 1 objects.

   The name views::maybe denotes a customization point object (??). Given a subexpression E, the expression views::maybe(E) is expression-equivalent to maybe_view<decay_t<decltype((E))>>(E).

   **Example 1:**
   ```cpp
   int i[4];
   for (int i : views::maybe(4))
       cout << i;  // prints 4
   
   maybe_view<int> m2[];
   for (int k : m2)
       cout << k;  // Does not execute
   ```

### 1.2 Class template maybe_view

```cpp
template <typename Value>
class maybe_view;

template <typename 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(const Value& value) requires copy_constructible<T>;
    constexpr explicit maybe_view(Value&& value);
    template <class... Args>
    requires constructible_from<T, Args...>
    constexpr maybe_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 maybe_view& lhs, const maybe_view& rhs) {
  return lhs.value_ <=> rhs.value_;}
}

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

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

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

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

constexpr explicit maybe_view(Value const& maybe);

Effects: Initializes value_ with maybe.

constexpr explicit maybe_view(Value&& maybe);

Effects: Initializes value_ with std::move(maybe).

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

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

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

Effects: Equivalent to: return data();

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

Returns: data() + size().

static constexpr size_t size() noexcept;

Effects: Equivalent to:
  return bool(value_);
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;

Returns: std::addressof(*value_);

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

Returns: lhs.value_ <=> rhs.value_;

constexpr auto operator==(const maybe_view& lhs, const maybe_view& rhs)

Returns: lhs.value_ == rhs.value_;

template <typename F>
constexpr auto and_then(F&& f) &;

Effects: Equivalent to:

using U = std::invoke_result_t<F, Value&>;
if (value_)
return std::invoke(std::forward<F>(f), *value_);
else {
return std::remove_cvref_t<U>();
}


template <typename F>
constexpr auto and_then(F&& f) &&;

Effects: Equivalent to:

using U = std::invoke_result_t<F, Value&&>;
if (value_)
return std::invoke(std::forward<F>(f), std::move(*value_));
else {
return std::remove_cvref_t<U>();
}


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

Effects: Equivalent to:

using U = std::invoke_result_t<F, const Value&>;
if (value_)
return std::invoke(std::forward<F>(f), *value_);
else {
return std::remove_cvref_t<U>();
}


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

Effects: Equivalent to:

using U = std::invoke_result_t<F, const Value&&>;
if (value_)
return std::invoke(std::forward<F>(f), std::move(*value_));
else {
return std::remove_cvref_t<U>();
}


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

Effects: Equivalent to:

using U = std::invoke_result_t<F, Value&>;
return (value_ ? maybe_view<U>{std::invoke(std::forward<F>(f), *value_)} \n
\n


\[
\begin{align*}
\text{template } & \text{ typename F} \\
& \text{ constexpr auto transform(F&& f) } \&; \\
\text{ Effects: } & \text{ Equivalent to:} \\
& \text{ using } U = \text{ std::invoke_result_t<F, Value&&>; } \\
& \text{ return (value_) ? maybe_view<U>{\text{std::invoke(std::forward<F>>(f), } } \\
& \text{ std::move(*value_)}} \\
& \text{ : maybe_view<U>{}; } \\
\end{align*}
\]

\[
\begin{align*}
\text{template } & \text{ typename F} \\
& \text{ constexpr auto transform(F&& f) const&; } \\
\text{ Effects: } & \text{ Equivalent to:} \\
& \text{ using } U = \text{ std::invoke_result_t<F, const Value&>; } \\
& \text{ return (value_) ? maybe_view<U>{\text{std::invoke(std::forward<F>>(f), *value_}) } \\
& \text{ : maybe_view<U>{}; } \\
\end{align*}
\]

\[
\begin{align*}
\text{template } & \text{ typename F} \\
& \text{ constexpr auto transform(F&& f) const&&; } \\
\text{ Effects: } & \text{ Equivalent to:} \\
& \text{ using } U = \text{ std::invoke_result_t<F, const Value&&>; } \\
& \text{ return (value_) ? maybe_view<U>{\text{std::invoke(std::forward<F>>(f), std::move(*value_))}} } \\
& \text{ : maybe_view<U>{}; } \\
\end{align*}
\]

\[
\begin{align*}
\text{template } & \text{ typename F} \\
& \text{ constexpr auto or_else(F&& f) } \&; \\
\text{ Effects: } & \text{ Equivalent to:} \\
& \text{ using } U = \text{ std::invoke_result_t<F>; } \\
& \text{ return value_ ? *this : std::forward<F>>(f()); } \\
\end{align*}
\]

\[
\begin{align*}
\text{template } & \text{ typename F} \\
& \text{ constexpr auto or_else(F&& f) const&; } \\
\text{ Effects: } & \text{ Equivalent to:} \\
& \text{ using } U = \text{ std::invoke_result_t<F>; } \\
& \text{ return value_ ? std::move(*this) : std::forward<F>>(f()); } \\
\end{align*}
\]

\[
\begin{align*}
\text{template } & \text{ typename Value} \\
& \text{ class maybe_view<Value&> : public ranges::view_interface<maybe_view<Value&>> } \\
& \text{ private:} \\
& \text{ Value* value_ ; // exposition only} \\
\text{ public:} \\
& \text{ constexpr maybe_view();} \\
& \text{ constexpr explicit maybe_view(Value& value);} \\
& \text{ constexpr explicit maybe_view(Value&& value) = delete;} \\
\end{align*}
\]
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 maybe_view& lhs, 
    const maybe_view& rhs);

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

template <typename F>
constexpr auto and_then(F&& f) &;

template <typename F>
constexpr auto and_then(F&& f) &&;

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

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

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

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

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

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

template <typename F>
constexpr maybe_view or_else(F&& f) &&;

constexpr explicit maybe_view();

Effects: Initializes value_ with nullptr

constexpr explicit maybe_view(Value maybe);

Effects: Initializes value_ with addressof(maybe)

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

Effects: Equivalent to: return data();

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

Returns: data() + size().

static constexpr size_t size() noexcept;

Effects: Equivalent to:

    return bool(value_);

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

Effects: Equivalent to:
   if (!value_
       return nullptr;
   return std::addressof(*value_);

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

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

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

Returns:
   (bool(lhs.value_ || bool(value_))
    ? (*lhs.value_ == *rhs.value_)
    : (bool(lhs.value_) == bool(rhs.value_));

template <typename F>
constexpr auto and_then(F& f) &;

Effects: Equivalent to:
   using U = std::invoke_result_t<F, Value&>;
   if (value_ { 
      return std::invoke(std::forward<F>(f), *value_); 
   } else { 
      return std::remove_cvref_t<U>(); 
   }

template <typename F>
constexpr auto and_then(F&& f) &&;

Effects: Equivalent to:
   using U = std::invoke_result_t<F, Value&&>;
   if (value_ { 
      return std::invoke(std::forward<F>(f), std::move(*value_)); 
   } else { 
      return std::remove_cvref_t<U>(); 
   }

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

Effects: Equivalent to:
   using U = std::invoke_result_t<F, const Value&>;
   if (value_ { 
      return std::invoke(std::forward<F>(f), *value_); 
   } else { 
      return std::remove_cvref_t<U>(); 
   }

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

Effects: Equivalent to:
  using U = std::invoke_result_t<F, const Value&&>;
  if (value_) {
    return std::invoke(std::forward<F>(f), std::move(*value_));
  } else {
    return std::remove_cvref_t<U>();
  }

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

Effects: Equivalent to:
  using U = std::invoke_result_t<F, Value&>;
  return (value_) ? maybe_view<U>{std::invoke(std::forward<F>(f), value_)}
  : maybe_view<U>();

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

Effects: Equivalent to:
  using U = std::invoke_result_t<F, Value&&>;
  return (value_) ? maybe_view<U>{std::invoke(std::forward<F>(f), std::move(*value_))}
  : maybe_view<U>();

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

Effects: Equivalent to:
  using U = std::invoke_result_t<F, const Value&>;
  return (value_) ? maybe_view<U>{std::invoke(std::forward<F>(f), *value_)}
  : maybe_view<U>();

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

Effects: Equivalent to:
  using U = std::invoke_result_t<F, const Value&&>;
  return (value_) ? maybe_view<U>{std::invoke(std::forward<F>(f), std::move(*value_))}
  : maybe_view<U>();

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

Effects: Equivalent to:
  using U = std::invoke_result_t<F>; return value_ ? *this : std::forward<F>(f());

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

Effects: Equivalent to:
using U = std::invoke_result_t<F>;
return value_ ? std::move(*this) : std::forward<F>(f)();

2.2 Nullable View
2.2.1 Overview
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_val or nullable_object_ref.

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).

2.2.2 Class template nullable_view
[range.nullable.view]
[Example 1]:
std::optional o{4};
for (int k : nullable_view m{o})
  cout << k;  // prints 4

— end example]
template <typename Nullable>
requires(copyable_object<Nullable> &&
  (nullable_object_val<Nullable> || nullable_object_ref<Nullable>))
class nullable_view<Nullable>
  : public ranges::view_interface<nullable_view<Nullable>> {
private:
  using T = std::remove_reference_t<
    std::iter_reference_t<typename std::unwrap_reference_t<Nullable>>>;

  Nullable* value_;  

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;
};

constexpr explicit nullable_view();

Effects: Initializes value_ with nullptr

constexpr explicit nullable_view(Nullable nullable);

Effects: Initializes value_ with addressof(nullable)

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

Effects: Equivalent to: return data();

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

Returns: data() + size();

static constexpr size_t size() noexcept;

Effects: Equivalent to:
  if (!value_)
    return 0;
  Nullable& m = *value_;  
  if constexpr (is_reference_wrapper_v<Nullable>) {
    return bool(m.get());
  } else {
    return bool(m);
  }

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

Effects: Equivalent to:
  if (!value_)
    return nullptr;
const Nullable& m = *\exposid{value_};
if constexpr (is_reference_wrapper_v<Nullable>) {
    return m.get() ? std::addressof(*m.get()) : nullptr;
} else {
    return m ? std::addressof(*m) : nullptr;
}

3 Feature-test macro

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_maybe 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 [N4910].

Document history

— 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


