Give `std::optional` Range Support

https://wg21.link/p3168r0

Marco Foco <marco.foco@gmail.com>
Darius Neațu <dariusn@adobe.com>
Barry Revzin <barry.revzin@gmail.com>
David Sankel <dsankel@adobe.com>

2024-03-20 Tokyo WG21 Meeting

Background

`std::optional`

A proposal to add a utility class to represent optional objects (N3793), Fernando Cacciola and Andrezej Krzemieński

- `std::optional<T>` is a class template that "may or may not store a value of type `T` in its storage space"
- Incorporated into C++17
- Frequent use in data members, function parameters, function return types, and stack variables.

Ranges

The One Ranges Proposal (P0896R4), Eric Niebler

- Concepts and algorithms
- Goal: simplify algorithm composition and usage

Usage experience

- Applying range and iteration algorithms to `std::optional` is a recurring topic in forums
- Optional types with range support are showing up in the wild (e.g. `owi::optional`)
- Other programming languages (e.g. Rust and Scala) treat optional as a range for algorithmic purposes

`views::maybe`

A view of 0 or 1 elements: `views::maybe` (P1255R12), Steve Downey

- Includes many examples illustrating utility of optional being a range
- Proposed two facilities:
  1. `views::nullable`. Range adapter producing view of `std::optional` and `std::optional`-like objects.
  2. `views::maybe`. Data type with `std::optional` semantics, with interface differences including range support.

Interface differences 1/2

`std::optional<V>`	both	`std::maybe_view<V>`
`Opt(nullopt)`	`Opt()`	
	`Opt(v)`	
	`Opt(in_place, v)`	
`o = nullopt;`	`o = v;`	
`o.emplace(v);`		
`o.reset();`		
`o.swap(o2);`		

Interface differences 2/2

`std::optional<V>`	**both**	`std::maybe_view<V>`
	`o == o2` and `o <=> o2`	
	`o == v` and `o <=> v`	
`*o` and `o->m`	`o.transform(f)`	`o.begin()` and `o.end()`
`o.has_value()` and `bool(o)`	`o.and_then(f)`	`o.size()`
`o.value_or(v)`	`o.or_else(f)`	`o.data()`
`std::hash<Opt>{}(o)`		

Key design principles

1. Genericity
2. Simplicity

Genericity 1/2

Fundamentals of Generic Programming, Stepanov and Dehnert

Generic programming recognizes that dramatic productivity improvements must come from reuse without modification, as with the successful libraries. Breadth of use, however, must come from the separation of underlying data types, data structures, and algorithms, allowing users to combine components of each sort from either the library or their own code. Accomplishing this requires more than just simple, abstract interfaces – it requires that a wide variety of components share the same interface so that they can be substituted for one another.

Genericity 2/2

It is vital that we go beyond the old library model of reusing identical interfaces with pre-determined types, to one which identifies the minimal requirements on interfaces and allows reuse by similar interfaces which meet those requirements but may differ quite widely otherwise. Sharing similar interfaces across a wide variety of components requires careful identification and abstraction of the patterns of use in many programs, as well as development of techniques for effectively mapping one interface to another.

Simplicity

Direction for ISO C++ (P2000R4), Direction Group

C++ in danger of losing coherency due to proposals based on differing and sometimes mutually contradictory design philosophies and differing stylistic tastes.

Revisiting `std::maybe`

`std::maybe_view` lacks dereference and bool conversion

- Contrary to simplicity & genericity
- These are standard for optional-like objects
- The interface originates in nullable C pointers
- Consider a generic serializer that outputs both a `vector<optional<T>>` and a `vector<shared_ptr<T>>`, but will not work with a `vector<maybe_view<T>`.

`std::maybe_view` contains lacks some accessors

- `transform`, `and_then`, `or_else` included, but not `value_or`
- `value_or` is very popular
- What is the basis for these decisions?

`std::maybe_view` satisfies the `range` concept

- `std::maybe_view` can be used in more algorithms. Great!
- But, this will confuse users with an otherwise-identical types

When would one choose `std::maybe_view` over `std::optional`?

When "the value will have operations applied if present, and ignored otherwise."

- For return types, how can one know ahead of time what the callers will do?
- For argument types, what if the implementation changes?

Let's not force users to spend mental energy like this!

Our proposal

Instead of introducing a new type, make `std::optional` satisfy the `ranges::range` concept where iterating over a `std::optional` object will iterate over its 0 or 1 elements.

An example

```
// A person's attributes (e.g., eye color). All attributes are optional.
class Person {
    /* ... */
public:
    optional<string> eye_color() const;
};

vector<Person> people = ...;
```

```
// Compute eye colors of 'people'.
vector<string> eye_colors = people
  | views::transform(&Person::eye_color)
  | views::transform(views::nullable)
  | views::join
  | ranges::to<set>()
  | ranges::to<vector>();
```

```
// Compute eye colors of 'people'.
vector<string> eye_colors = people
  | views::transform(&Person::eye_color)
  // no extra wrapping necessary
  | views::join
  | ranges::to<set>()
  | ranges::to<vector>();
```

Every example from `views::maybe` gets simpler

Design particulars (rationale in paper)

- Add `[c][r]{begin|end}` family of member functions
- Specialize `ranges::enable_view`
- `T*` as iterator