Document #: P2439R0
Date: 2021-09-09
Project: Programming Language C++
Audience: LEWG
Reply-to: Barry Revzin <barry.revzin@gmail.com>
Tim Song <t.canens.cpp@gmail.com>

What is a `view`?

P2415R1

`view`, a history

N4128, *Ranges for the Standard Library* (2014)

- First proposed (as Range)
  - "lightweight objects that denote a range of elements they do not own"
  - O(1) copyable and assignable, default constructible

Views are:

- Non-owning
- O(1) default constructible
- O(1) copy constructible
- O(1) copy assignable
- O(1) move constructible
- O(1) move assignable

`view`, a history

P0789, *Range Adaptors and Utilities* (2017)

- Proposed `single_view` – an owning view

Views are:

- ~~Non-owning~~
- O(1) default constructible
- O(1) copy constructible
- O(1) copy assignable
- O(1) move constructible
- O(1) move assignable

`view`, a history

P1456, *Move-only views* (2019)

- Copyability no longer required
- But copy operations must be O(1) where supported
- Destruction required to be O(1)

Views are:

- O(1) default constructible
- O(1) copy constructible if supported
- O(1) copy assignable if supported
- O(1) move constructible
- O(1) move assignable
- O(1) destructible

`view`, a history

P2325, *Views should not be required to be default constructible* (2021)

- Default constructible requirement removed

Views are:

- ~~O(1) default constructible~~
- O(1) copy constructible if supported
- O(1) copy assignable if supported
- O(1) move constructible
- O(1) move assignable
- O(1) destructible

Why does `view` have complexity requirements?

Look at the algorithms – range adaptors *are* the algorithm for views:

```
auto rng = some_view
         | views::reverse
         | views::take(42)
         | views::transform(f);
```

This pipeline:

- Copies `some_view` once
- Moves it ~5 times
- Destroys all the moved-from temporaries

Complexity requirements exist to support efficient lazy composition of views.

What do the algorithms actually need?

```
struct bad_view : view_interface<bad_view>
{

    std::vector<std::string> v;

    bad_view(std::vector<std::string> v)
      : v(std::move(v)) { }

    auto begin() { return v.begin(); }

    auto end()   { return v.end(); }

};

std::vector<std::string> get();
```

`bad_view`

- O(1) move constructible
- Copyable, but not O(1) copyable
- Not O(1) destructible

What breaks when it is used as a view?

- `auto rng = bad_view(get())`
  `| views::enumerate;`
  - OK, pipeline constructed in constant time
- `auto bv = bad_view(get());`
  `auto rng = bv | views::enumerate;`
  - Construction of `rng` copies `bv`

What do the algorithms actually need?

```
struct bad_view2 : view_interface<bad_view2>
{

    std::vector<std::string> v;

    bad_view2(std::vector<std::string> v)
      : v(std::move(v)) { }

    bad_view2(bad_view2&&) = default;

    bad_view2& operator=(bad_view2&&) = default;

    auto begin() { return v.begin(); }

    auto end()   { return v.end(); }

};

std::vector<std::string> get();
```

`bad_view2`

- O(1) move constructible
- Not copyable
- Not O(1) destructible

What breaks when it is used as a view?

- ```
  auto rng = bad_view2(get())
           | views::enumerate;
  ```
  - Still OK – constant time
- ```
  auto bv = bad_view2(get());
  auto rng = bv | views::enumerate;
  ```
  - Ill-formed – `bad_view2` is not copyable

Writing the **bad_view2** example today

```
auto strings = get();
auto rng = strings | views::enumerate;
```

Doesn't move the vector – but move construction is cheap.

`rng` holds a reference to `strings` – extra indirection, risk of dangling

Destruction of `rng` is O(1)…but we still have to destroy `strings` anyway and pay the cost there

Proposal: relax complexity requirements

T models `view` only if:

- T has O(1) move construction; and
- ~~T has O(1) move assignment~~ move assignment of an object of type T is no more complex than destruction followed by move construction; and
- ~~T has O(1) destruction~~ if *N* copies and/or moves are made from an object of type T that contained *M* elements, then those *N* objects have *O(N+M)* destruction; and
- `copy_constructible<T>` is false, or T has O(1) copy construction; and
- `copyable<T>` is false, or ~~T has O(1) copy assignment~~ copy assignment of an object of type T is no more complex than destruction followed by copy construction.

Proposal: auto-wrapping non-views

Add a move-only `owning_view` adaptor:

```
template<range R>
   requires /* … */
 class owning_view;
```

Change `views::all` wrap rvalue non-views with `owning_view`, enabling such types to be used in view adaptor pipelines.

Update `viewable_range` to match `views::all`.

Example (ill-formed today, valid with this change):

```
std::vector<int> get_ints();

auto rng = get_ints()
         | views::filter(pred)
         | views::transform(f);
```

What is a `view`?

```
auto rng = v | views::reverse;
```

Should `rng` store a *copy* of `v` or a *reference* to it?

- If it should store a copy because copying is cheap and it's better to avoid potential dangling and cost of indirection, `v` is a `view`.
- If it should store a reference to `v` because copying is expensive, `v` is not a `view`.