

Optimize for std::optional in range adaptors

Steve Downey <sdowney@gmail.com>
Tomasz Kamiński <tomaszkam@gmail.com>

Document #: P3913R0
Date: 2025-11-05
Project: Programming Language C++
Audience: LEWG,LWG

Abstract

From PL-011 22.5 [optional] Optimize for std::optional in range adaptors

The range support was added to the optional, making it usable with range adaptors defined in std::views, however, we have not updated the views specification to handle it optimally when possible. This leads to unnecessary template instantiations.

Contents

1	Motivation	1
2	Design	1
3	Wording	2
	References	4

1 Motivation

The range support was added to the optional, making it usable with range adaptors defined in std::views, however, we have not updated the views specification to handle it optimally when possible. This leads to unnecessary template instantiations.

Proposed change:

Add a special case to recognize optional for adaptors:

views::as_const: should return optional or optional<const U&> (if T is U&) views::take(opt, n): empty optional if n is equal to zero, opt otherwise views::drop(opt, n): empty optional if n greater than zero, opt otherwise views::reverse: input unchanged

2 Design

2.1 views::as_const

return optional or optional<const U&> (if T is U&)

2.2 views::take(opt, n)

empty optional if n is equal to zero, opt otherwise

2.3 views::drop(opt, n)

empty optional if n greater than zero, opt otherwise

2.4 views::reverse

input unchanged

3 Wording

The proposed changes are relative to the current working draft [N5014].

❖.1 General

[ranges.general]

❖.❖.1 Take view

[range.take]

❖.❖.1.1 Overview

[range.take.overview]

- ¹ `take_view` produces a view of the first N elements from another view, or all the elements if the adapted view contains fewer than N .
- ² The name `views::take` denotes a range adaptor object. Let E and F be expressions, let T be `remove_cvref_t<decltype((E))>`, and let D be `range_difference_t<decltype((E))>`. If `decltype((F))` does not model `convertible_to<D>`, `views::take(E, F)` is ill-formed. Otherwise, the expression `views::take(E, F)` is expression-equivalent to:
 - (2.1) — If T is a specialization of `empty_view`, then `((void)F, decay-copy(E))`, except that the evaluations of E and F are indeterminately sequenced.
 - (2.2) — Otherwise, If T is specialization of `optional`, then `(F == D(0) ? ((void)E, T()) : decay-copy(E))`
 - (2.3) — Otherwise, if T models `random_access_range` and `sized_range` and is a specialization of `span`, `basic_string_view`, or `subrange`, then `U(ranges::begin(E), ranges::begin(E) + std::min<D>(ranges::distance(E), F))`, except that E is evaluated only once, where U is a type determined as follows:
 - (2.3.1) — if T is a specialization of `span`, then U is `span<typename T::element_type>`;
 - (2.3.2) — otherwise, if T is a specialization of `basic_string_view`, then U is T ;
 - (2.3.3) — otherwise, T is a specialization of `subrange`, and U is `subrange<iterator_t<T>>`;
 - (2.4) — otherwise, if T is a specialization of `iota_view` that models `random_access_range` and `sized_range`, then `iota_view(*ranges::begin(E), *(ranges::begin(E) + std::min<D>(ranges::distance(E), F)))`, except that E is evaluated only once.
 - (2.5) — Otherwise, if T is a specialization of `repeat_view`:
 - (2.5.1) — if T models `sized_range`, then`views::repeat(*E.value_, std::min<D>(ranges::distance(E), F))`
except that E is evaluated only once;
 - (2.5.2) — otherwise, `views::repeat(*E.value_, static_cast<D>(F))`.
 - (2.6) — Otherwise, `take_view(E, F)`.

❖.❖.2 Drop view

[range.drop]

❖.❖.2.1 Overview

[range.drop.overview]

- ¹ `drop_view` produces a view excluding the first N elements from another view, or an empty range if the adapted view contains fewer than N elements.
- ² The name `views::drop` denotes a range adaptor object. Let E and F be expressions, let T be `remove_cvref_t<decltype((E))>`, and let D be `range_difference_t<decltype((E))>`. If `decltype((F))` does not model `convertible_to<D>`, `views::drop(E, F)` is ill-formed. Otherwise, the expression `views::drop(E, F)` is expression-equivalent to:
 - (2.1) — If T is a specialization of `empty_view`, then `((void)F, decay-copy(E))`, except that the evaluations of E and F are indeterminately sequenced.
 - (2.2) — Otherwise, If T is specialization of `optional`, then `(F == D(0) ? decay-copy(E) : ((void)E, T()))`

- (2.3) — Otherwise, if T models `random_access_range` and `sized_range` and is
 - (2.3.1) — a specialization of `span`,
 - (2.3.2) — a specialization of `basic_string_view`,
 - (2.3.3) — a specialization of `iota_view`, or
 - (2.3.4) — a specialization of `subrange` where `T::StoreSize` is false,
 then `U(ranges::begin(E) + std::min<D>(ranges::distance(E), F), ranges::end(E))`, except that E is evaluated only once, where U is `span<typename T::element_type>` if T is a specialization of `span` and T otherwise.
- (2.4) — Otherwise, if T is a specialization of `subrange` that models `random_access_range` and `sized_range`, then
 `T(ranges::begin(E) + std::min<D>(ranges::distance(E), F), ranges::end(E), to-unsigned-like(ranges::distance(E) - std::min<D>(ranges::distance(E), F)))`, except that E and F are each evaluated only once.
- (2.5) — Otherwise, if T is a specialization of `repeat_view`:
 - (2.5.1) — if T models `sized_range`, then
 `views::repeat(*E.value_, ranges::distance(E) - std::min<D>(ranges::distance(E), F))`
 except that E is evaluated only once;
 - (2.5.2) — otherwise, `((void)F, decay-copy(E))`, except that the evaluations of E and F are indeterminately sequenced.
- (2.6) — Otherwise, `drop_view(E, F)`.

❖.❖.3 As const view

[range.as.const]

❖.❖.3.1 Overview

[range.as.const.overview]

- ¹ `as_const_view` presents a view of an underlying sequence as constant. That is, the elements of an `as_const_view` cannot be modified.
- ² The name `views::as_const` denotes a range adaptor object. Let E be an expression, let T be `decltype((E))`, and let U be `remove_cvref_t<T>`. The expression `views::as_const(E)` is expression-equivalent to:
 - (2.1) — If `views::all_t<T>` models `constant_range`, then `views::all(E)`.
 - (2.2) — Otherwise, if U denotes `empty_view<X>` for some type X, then `auto(views::empty<const X>)`.
 - (2.3) — Otherwise, if U denotes `optional<X&>` for some type X, then `optional<const X&>(E)`
 - (2.4) — Otherwise, if U denotes `optional<X>` for some type X, then `optional<const X>(E)`
 - (2.5) — Otherwise, if U denotes `span<X, Extent>` for some type X and some extent Extent, then `span<const X, Extent>(E)`.
 - (2.6) — Otherwise, if U denotes `ref_view<X>` for some type X and `const X` models `constant_range`, then `ref_view(static_cast<const X&>(E.base()))`.
 - (2.7) — Otherwise, if E is an lvalue, `const U` models `constant_range`, and U does not model view, then `ref_view(static_cast<const U&>(E))`.
 - (2.8) — Otherwise, `as_const_view(E)`.

❖.❖.4 Reverse view

[range.reverse]

❖.❖.4.1 Overview

[range.reverse.overview]

- ¹ `reverse_view` takes a bidirectional view and produces another view that iterates the same elements in reverse order.
- ² The name `views::reverse` denotes a range adaptor object. Given a subexpression E, the expression `views::reverse(E)` is expression-equivalent to:
 - (2.1) — If the type of E is a (possibly cv-qualified) specialization of `reverse_view`, then `E.base()`.
 - (2.2) — Otherwise, If T is specialization of `optional`, then `decay-copy(E)`.
 - (2.3) — Otherwise, if the type of E is `cv subrange<reverse_iterator<I>, reverse_iterator<I>, K>` for some iterator type I and value K of type `subrange_kind`,

(2.3.1) — if `K` is `subrange_kind::sized`, then `subrange<I, I, K>(E.end().base(), E.begin().base(), E.size());`

(2.3.2) — otherwise, `subrange<I, I, K>(E.end().base(), E.begin().base())`.

However, in either case `E` is evaluated only once.

(2.4) — Otherwise, `reverse_view{E}`.

References

[N5014] Thomas Köppe. N5014: Working draft, standard for programming language c++. <https://wg21.link/n5014>, 8 2025.