

views::trim and views::trim_while

Document #: P4398R0 [Latest] [Status]
Date: 2026-10-19
Project: ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++
Audience: SG9 Ranges
LEWG
Reply-to: Anish Aggarwal
<anishagg17@gmail.com>

Contents

1	Abstract	1
2	Revision history	2
2.1	R0 (2026-10-19)	2
3	Motivation	2
3.1	The operation	2
3.2	Examples	2
3.3	Why not compose existing adaptors?	3
3.4	Use in other languages and libraries	3
3.5	Scope	3
3.6	Impact on the standard	4
4	Design	4
4.1	Predicate direction	4
4.2	Names: <code>trim_while</code> and <code>trim</code>	4
4.3	Requirements: bidirectional and common	4
4.4	Mechanism	4
4.5	<code>sized_range</code>	5
4.6	<code>constexpr</code>	5
4.7	No projection	5
4.8	Alternatives considered	5
4.9	Questions for SG9	5
5	Implementation experience	5
6	Wording	6
6.1	25.2 Header <code><ranges></code> synopsis [ranges.syn]	6
6.2	[range.trim]	7
6.3	17.3.2 Header <code><version></code> synopsis [version.syn]	8
7	Acknowledgements	8
8	References	9

1 Abstract

[P2760R1] lists `views::trim` as a Tier 2 range adaptor to bring over from range-v3. As far as I can find, it has not been proposed yet. This paper proposes two range adaptors in `<ranges>`:

- `views::trim_while(pred)`: drop the longest leading run and the longest trailing run of elements for which `pred` is `true`;
- `views::trim(value)`: the same, for elements equal to `value`.

```
std::string_view s = "  hello  ";
auto t = s | std::views::trim(' ');           // "hello"

std::vector<int> v = {0, 0, 7, 8, 0};
auto u = v | std::views::trim_while([](int i) { return i == 0; }); // {7, 8}
```

Both are backed by one new view, `trim_view`. It needs a bidirectional, common range, and its iterators are the base range's own iterators, so the result keeps the base's iterator category (including contiguity) and is always a common range. The semantics are range-v3's `views::trim`. The predicate form's name and the `sized_range` behaviour differ from range-v3 on purpose; both are put to SG9 as questions.

2 Revision history

2.1 R0 (2026-10-19)

First numbered revision, for the 2026-10 mailing. Changes since the draft posted to std-proposals on 2026-10-13:

- The predicate form was renamed from `trim_if` to `trim_while`, following feedback on the list ([§ Design](#)).
- The reference implementation was fixed in several places found in review: cached iterators are no longer copied or moved with the view, the predicate is stored so that capturing lambdas work, and closure composition works in C++23. No change to the proposed behaviour.

3 Motivation

3.1 The operation

Removing padding from both ends of a sequence is common: whitespace around a token, zero samples around a signal, sentinel values around a payload, empty slots around the live part of a buffer. C++ has `views::drop_while` for the front, but nothing for both ends.

3.2 Examples

Trimming a token. Today, the usual way is the `string_view` members:

```
std::string_view trim_ws(std::string_view s) {
    auto first = s.find_first_not_of(" \t\n\r");
    if (first == std::string_view::npos) return {};
    auto last = s.find_last_not_of(" \t\n\r");
    return s.substr(first, last - first + 1);
}
```

With this paper:

```
auto is_ws = [](char c) { return c == ' ' || c == '\t' || c == '\n' || c == '\r'; };
// The result is contiguous, so it converts back to a string_view.
std::string_view trimmed(s | std::views::trim_while(is_ws));
```

The hand-written version has two traps: the all-whitespace case (without the `npos` check, `last - first + 1` is computed from two `npos` values), and the `+ 1`, because `find_last_not_of` returns the position of the last *kept* character rather than one past it. It also only works for strings: for any other range, the same job needs `ranges::find_if_not`, C++23's `ranges::find_last_if_not`, and the same two checks.

Trimming silence from audio samples. A signal buffer padded with zeros:

```
void analyse(std::span<const float> samples) {
    for (float x : samples | std::views::trim(0.0f)) {
        // only the part between the first and last non-zero sample
    }
}
```

Because the result’s iterators are the `span`’s own iterators, the kept part can be turned back into a `std::span` directly:

```
auto kept = samples | std::views::trim(0.0f);
std::span<const float> body(kept.begin(), kept.end());
```

Composing. Trimming is often one step of a pipeline:

```
auto fields = line
    | std::views::split(',')
    | std::views::transform([](auto f) {
        return std::string_view(f | std::views::trim(' '));
    });
```

3.3 Why not compose existing adaptors?

- `views::drop_while(pred)` removes the prefix only.
- `rng | views::reverse | views::drop_while(pred) | views::reverse` handles the suffix and is lazy, but the iterators become `reverse_iterator<reverse_iterator<I>>`. The result is not a `contiguous_range` even when `rng` is one (checked with `libstdc++`: random access, sized and common are kept; contiguity is lost), so it cannot be turned back into a `string_view` or `span` without unwrapping `.base()` twice.
- `ranges::find_last_if_not` (C++23) finds the trailing boundary but returns an algorithm result, not a view, and the caller still has to assemble the kept range by hand.

`range-v3`’s own test for `trim` [range-v3] compares it against

```
ia | views::drop_while(p) | views::reverse | views::drop_while(p)
```

That check only works because its expected result, `{4, 3, 2, 1, 2, 3, 4}`, is a palindrome; the composition produces the elements in reverse order.

3.4 Use in other languages and libraries

Trimming both ends is a standard operation elsewhere, in both forms this paper proposes:

Library	Predicate form	Value or character-set form
range-v3	<code>views::trim(pred)</code>	—
Boost.StringAlgo	<code>trim_if(seq, pred)</code>	<code>trim(seq)</code> (whitespace)
Python	—	<code>str.strip(chars)</code> ; NumPy <code>trim_zeros(a)</code>
Rust	<code>str::trim_matches(c ...)</code>	<code>str::trim_matches(c)</code>
.NET	—	<code>MemoryExtensions.Trim</code> on spans
Kotlin	<code>CharSequence.trim(predicate)</code>	<code>String.trim(vararg chars)</code>
Haskell	<code>Data.Text.dropAround(pred)</code>	—

The .NET and NumPy entries matter most here: they are generic over element type (not just characters), which is what `views::trim(value)` offers.

I have not yet measured how often hand-written trimming of non-string ranges appears in C++ code. Hewill Kang asked for this kind of evidence on `std`-proposals for the related `drop_last_while` proposal, and the same question applies here. I plan to add a code-search survey in R1.

3.5 Scope

In scope: `trim_view`, `views::trim_while`, `views::trim`, a feature-test macro.

Not in scope: a mutating `ranges::trim` algorithm; a “set of values” form (Python’s `strip(chars)`); support for forward-only ranges; string member functions; `views::drop_last_while` and `views::take_last_while` (P2760 Tier 1, see [alternatives](#)).

3.6 Impact on the standard

A pure library addition to `<ranges>`: one new class template, two range adaptor objects, an `enable_borrowed_range` specialisation and a feature-test macro. No existing wording changes, nothing is deprecated, and no core language change is needed. It can be implemented in C++20 on top of the existing ranges machinery.

4 Design

4.1 Predicate direction

`pred(e) == true` means *e is removed*, as in `drop_while`, `remove_if`, Boost’s `trim_if` and `range-v3`.

4.2 Names: `trim_while` and `trim`

`range-v3` uses one name, `views::trim(pred)`. This paper uses two: `trim_while(pred)` for a predicate and `trim(value)` for a value. The split mirrors the plan’s own Tier 1 pairs `drop_last` / `drop_last_while` and `take_last` / `take_last_while`, and the library’s `remove` / `remove_if`. Separate names also avoid an ambiguity a single overloaded `trim(x)` would have: if the elements are themselves callable (function pointers, say), `x` could be read either as a predicate or as a value to compare against.

Why does `trim` take a value when `drop` and `take` take a count? Because a count is not something a caller of `trim` knows: trimming is defined by what the padding is, and the amount is what the scan finds. A caller who knows the counts already has `views::drop` and the proposed `views::drop_last` [P4294R0].

The draft posted to `std-proposals` called the predicate form `trim_if`. Arthur O’Dwyer replied that “`trim_while`” “would be the better and more appropriate/consistent name” [ODwyer]. I agree: the predicate is a “while” condition, as in `drop_while`, not a per-element test as in `remove_if`.

The value form earns its place because it is the common case: trimming spaces, zeros, NUL bytes or sentinel values. `trim(' ')` reads better than the equivalent lambda.

A third form, trimming any element of a *set* of values (Python’s `str.strip(chars)`, or C#’s `String.Trim` taking several characters), is not proposed. For an arbitrary element type it raises questions the predicate form already answers: what counts as a set, and how is it searched? It can be added later without changing either proposed form.

4.3 Requirements: `bidirectional` and `common`

`trim_view` requires `bidirectional_range<V>` and `common_range<V>` (as `range-v3` does), and the same predicate requirements as `drop_while_view`:

```
is_object_v<Pred> && indirect_unary_predicate<const Pred, iterator_t<V>>
```

- *Bidirectional*: the trailing bound is found by walking back from the end. A forward-only range cannot be trimmed lazily at the back, and quietly trimming only the front would hide the problem. Forward ranges are rejected at compile time; `drop_while` already covers their front.
- *Common*: the backward walk starts from an iterator at the end, and a sentinel cannot be decremented. `views::common` fixes this for users.

4.4 Mechanism

The view stores the base and the predicate (in a *movable-box*, as `drop_while_view` does, so a capturing lambda still gives a `view`). `begin()` is the first element not matching `pred`. `end()` is one past the last element not matching `pred`, found by walking back from the end and never passing `begin()`. Each is computed once and cached in a *non-propagating-cache*, as `drop_while_view` caches `begin()`. Copying or moving the view does not carry the cache, because a cached iterator may point into the view object itself (for example with `views::single`).

Because the bounds are base iterators, there is no iterator wrapper:

Property	<code>trim_view<V, Pred></code>
<code>view</code> , <code>common_range</code>	always
<code>bidirectional_range</code>	always (required)
<code>random_access_range</code> , <code>contiguous_range</code>	iff <code>V</code> is
<code>borrowed_range</code>	iff <code>V</code> is
<code>sized_range</code>	iff <code>V</code> 's iterators are a sized-sentinel pair

The first `begin()` scans the leading run; the first `end()` scans the trailing run. Later calls are $O(1)$.

There is no `const begin()` or `end()`. A `const` overload could not use the cache, so it would rescan on every call, which breaks the amortized constant time required of `begin()`. `drop_while_view` makes the same choice.

4.5 sized_range

range-v3's `trim_view` is never a `sized_range`. This paper follows `drop_while_view` instead: when the base's iterators are a sized-sentinel pair, `ranges::size` works through `end() - begin()` and gives the right answer. The first `size()` call pays for both scans. (Specialising `disable_sized_range` to `true`, as range-v3 does, would not change this: `ranges::size` still falls back to `end() - begin()` for a sized-sentinel pair. Making the view never sized would need a distinct sentinel type, which would also give up `common_range`.)

4.6 constexpr

All members of `trim_view` and both adaptors' call operators are `constexpr`, so trimming works in constant expressions wherever the base range does. The tests check this over `std::array`.

4.7 No projection

range-v3's `trim` takes an optional projection. `drop_while`, `take_while` and `filter` do not, so neither does this paper; `views::transform` covers it.

4.8 Alternatives considered

- **One name for both forms (range-v3's `trim(pred)`):** possible; see question 1.
- **A `ranges::trim` algorithm returning a `subrange`:** reasonable as a separate paper; the plan item is the adaptor.
- **Never sized, as range-v3:** needs a distinct sentinel type. See question 3.
- **Forward-range support by buffering:** a different, owning adaptor; not proposed.
- **Specify `trim_while` via `drop_last_while`:** `trim_while(p)` has the same elements as `drop_while(p)` followed by `drop_last_while(p)`. The latter is a P2760 Tier 1 item that is not proposed yet. If it is added later, `trim_view` could be specified in terms of it without changing behaviour. A dedicated view is proposed here because it keeps the base's own iterators.

4.9 Questions for SG9

1. Two names (`trim_while(pred)`, `trim(value)`), or range-v3's single `trim(pred)`?
2. Require bidirectional and common, with no forward-only mode?
3. Follow `drop_while_view` for `sized_range`, rather than range-v3?
4. Should a set-of-values form or a projection be added?
5. Should `common_range` be relaxed by making the range common internally, at the cost of a wrapper iterator?
6. A new `__cpp_lib_ranges_trim` macro, or a bump of `__cpp_lib_ranges`?
7. Forward to LEWG for C++29?

5 Implementation experience

A reference implementation and tests are at [\[impl\]](#). It is header-only, in namespace `std_proposal`, and follows the wording below. The script that runs every configuration below, and its full output, are in the

`verification/` folder of the same repository.

Tested on 2026-10-19 (GCC and Clang packages from Ubuntu 24.04):

Compiler	Library	Standards	Result
GCC 13.3	libstdc++ 13	C++20, C++23	pass
GCC 14.2	libstdc++ 14	C++20, C++23,	pass
		C++2c; ASan + UBSan	
Clang 18.1	libstdc++ 13	C++20, C++23,	pass
		C++2c	
Clang 20.1	libc++ 20	C++2c	pass
Clang 20.1	libstdc++ 14	C++20 pass; C++23/2c	
		see below	
MSVC	MS STL	—	not yet tested

With Clang 20 and libstdc++ 14 in C++23 mode, one test does not compile: composing two closures before applying them, as in

```
auto c = views::trim(' ') | views::reverse;
auto r = s | c;
```

A small example in `verification/` that uses none of this paper’s code fails in the same way with that combination, and compiles with GCC 14, with Clang 20 and libc++, and with Clang 18 and libstdc++ 13. All other tests pass with Clang 20 and libstdc++ 14.

The tests cover empty ranges, all elements trimmed, none trimmed, single elements, interior matches being kept, `const` element access, `constexpr` use, `borrowed_range` propagation, the iterator concept for `vector`, `deque` and `list` bases, the return types in the wording, the explicit constructor, rejection of forward-only and non-common bases and of non-comparable values, and the number of predicate calls (the first `begin()` makes one call per leading element plus one; later calls make none; the backward scan never passes `begin()`).

Building the implementation settled three points of the wording: the default constructor must be constrained (unconstrained, it is a hard error with `ref_view` bases under Clang); the predicate needs a *movable-box* (without it, a capturing lambda makes `trim_view` fail to model `view`); and the caches must be non-propagating on both copy and move.

No performance measurements have been made.

6 Wording

The wording is relative to [N5054]. Section numbers follow the current draft and may need updating. The feature-test macro value is a placeholder.

6.1 25.2 Header `<ranges>` synopsis [ranges.syn]

In 25.2 Header `<ranges>` synopsis [ranges.syn], add after the declarations for `drop_while_view`:

```
// [range.trim], trim view
template<view V, class Pred>
    requires bidirectional_range<V> && common_range<V> && is_object_v<Pred> &&
        indirect_unary_predicate<const Pred, iterator_t<V>>
class trim_view;

template<class T, class Pred>
    constexpr bool enable_borrowed_range<trim_view<T, Pred>> = enable_borrowed_range<T>;

namespace views {
    inline constexpr unspecified trim_while = unspecified;
    inline constexpr unspecified trim = unspecified;
}
```

6.2 [range.trim]

Add a new subclause after 25.7.13 Drop while view [range.drop.while]:

25.7.? Trim view [range.trim]

25.7.?.1 Overview [range.trim.overview]

- ¹ Given a unary predicate `pred` and a view `r`, `trim_view` produces a view of the elements of `r` with the longest leading sequence of elements satisfying `pred` and the longest trailing sequence of elements satisfying `pred` removed.
- ² The name `views::trim_while` denotes a range adaptor object (25.7.2 Range adaptor objects [range.adaptor.object]). Given subexpressions `E` and `F`, the expression `views::trim_while(E, F)` is expression-equivalent to `trim_view(E, F)`.
- ³ The name `views::trim` denotes a range adaptor object (25.7.2 Range adaptor objects [range.adaptor.object]). Given subexpressions `E` and `X`, the expression `views::trim(E, X)` is expression-equivalent to `trim_view(E, trim-equal-to<decay_t<decltype((X))>>{X})`

where *trim-equal-to* is the following exposition-only class template:

```
namespace std::ranges {
    template<class T>
    struct trim-equal-to {                                // exposition only
        T value;

        template<class U>
        requires requires(U&& u, const T& v) {
            { std::forward<U>(u) == v } -> boolean-testable;
        }
        constexpr bool operator()(U&& u) const {
            return bool(std::forward<U>(u) == value);
        }
    };
}
```

[Drafting note: Because `operator()` is constrained, a non-comparable element type makes the adaptor call ill-formed at the adaptor, rather than causing an error inside `trim_view`. Storing a copy of `X` means a closure such as `views::trim(' ')` does not dangle when it outlives the full-expression. The exact form is for LWG.]

- ⁴ [Example 1:

```
std::string_view s = " hello ";
for (char c : s | views::trim(' ')) {
    cout << c;    // prints hello
}
```

— end example]

25.7.?.2 Class template `trim_view` [range.trim.view]

```
namespace std::ranges {
    template<view V, class Pred>
    requires bidirectional_range<V> && common_range<V> && is_object_v<Pred> &&
        indirect_unary_predicate<const Pred, iterator_t<V>>
    class trim_view : public view_interface<trim_view<V, Pred>> {
    public:
        trim_view() requires default_initializable<V> && default_initializable<Pred>
            = default;
        constexpr explicit trim_view(V base, Pred pred);
    };
```

```

constexpr V base() const & requires copy_constructible<V> { return base_; }
constexpr V base() && { return std::move(base_); }

constexpr const Pred& pred() const;

constexpr iterator_t<V> begin();
constexpr iterator_t<V> end();

private:
    V base_ = V(); // exposition only
    movable_box<Pred> pred_; // exposition only
    non-propagating-cache<iterator_t<V>> begin_; // exposition only
    non-propagating-cache<iterator_t<V>> end_; // exposition only
};

template<class R, class Pred>
    trim_view(R&&, Pred) -> trim_view<views::all_t<R>, Pred>;
}

```

```
constexpr explicit trim_view(V base, Pred pred);
```

¹ *Effects:* Initializes `base_` with `std::move(base)` and `pred_` with `std::move(pred)`.

```
constexpr const Pred& pred() const;
```

² *Effects:* Equivalent to: `return *pred_;`

```
constexpr iterator_t<V> begin();
```

³ *Preconditions:* `pred_.has_value()` is true.

⁴ *Effects:* If `begin_` does not contain a value, initializes it with `ranges::find_if_not(base_, cref(*pred_))`.

⁵ *Returns:* `*begin_`.

```
constexpr iterator_t<V> end();
```

⁶ *Preconditions:* `pred_.has_value()` is true.

⁷ *Effects:* If `end_` does not contain a value, let `first` be `begin()` and let `last` be `ranges::end(base_)`. While `last != first` and `bool(invoke(*pred_, *ranges::prev(last)))` is true, decrements `last`. Then initializes `end_` with `last`.

⁸ *Returns:* `*end_`.

⁹ *Remarks:* In order to provide the amortized constant time complexity required by the `range` concept, the first calls of `begin()` and `end()` cache their results within the `trim_view` for use on subsequent calls.

[*Note 1:* Without this, applying a `reverse_view` over a `trim_view` would have quadratic iteration complexity. — *end note*]

6.3 17.3.2 Header `<version>` synopsis [version.syn]

In 17.3.2 Header `<version>` synopsis [version.syn], add after `__cpp_lib_ranges_to_container`:

```
#define __cpp_lib_ranges_trim 20XXXXL // freestanding, also in <ranges>
```

7 Acknowledgements

Thanks to Barry Revzin for the plan in [P2760R1], to Arthur O'Dwyer for the `trim_while` name (in his reply on the related `drop_last_while` thread), and to Johel Guerrero and the `range-v3` contributors for the design this builds on.

AI tools were used to help draft this paper and its reference implementation; the author has reviewed both.

8 References

- [impl] Anish Aggarwal. Reference implementation, tests and verification output for P4398.
<https://github.com/anishagg17/cpp-proposals/tree/main/views-trim>
- [N5054] Thomas Köppe. Working Draft, Programming Languages — C++.
<https://wg21.link/n5054>
- [ODwyer] Arthur O'Dwyer. 2026-10-14. Re: `views::drop_last_while` / `views::take_last_while` (std-proposals).
<https://lists.isocpp.org/std-proposals/2026/10/19200.php>
- [P2760R1] Barry Revzin. A Plan for C++26 Ranges.
<https://wg21.link/p2760r1>
- [P4294R0] Hewill Kang. `views::take_last` and `views::drop_last`.
<https://wg21.link/p4294r0>
- [range-v3] Eric Niebler. range-v3: `views::trim`.
<https://github.com/ericniebler/range-v3/blob/master/include/range/v3/view/trim.hpp>