# `ranges::to`: A function to convert any range to a container


| | |
|---|---|
| Document #: | P1206R3 |
| Date: | 2021-01-31 |
| Project: | Programming Language C++ |
| Audience: | LEWG |


| | |
|---|---|
| Reply-to: | Corentin Jabot <corentin.jabot@gmail.com> |
| | Eric Niebler <eric.niebler@gmail.com> |
| | Casey Carter <casey@carter.net> |


## Abstract


We propose a function to copy or materialize any range (containers and views alike) to a container.


## Revisions

### Revision 3

- Add support for `from_range_t`
- Add support for nested containers
- Remove syntax without parenthesis

### Revision 2

- Remove the implicit const removal when converting an associative container to a container of pairs
- Use CTAD to determine the value type of the returned container
- Attempt at wording

### Revision 1

- Split out the proposed constructors for string view and span into separate papers ([**?**] and [**?**] respectively)
- Use a function based approach rather than adding a constructor to standard containers, as it proved unworkable.

## Quick Overview

We propose all the following syntaxes to be valid constructs

```
std::list<int>  l;
std::map<int, int> m;

// copy a list to a vector of the same type
same_as<std::vector<int>> auto a = ranges::to<std::vector<int>>(l);
//Specify an allocator
same_as<std::vector<int, Alloc>> auto b = ranges::to<std::vector<int, Alloc>>(l, alloc);
// copy a list to a vector of the same type, deducing value_type
same_as<std::vector<int>> auto c = ranges::to<std::vector>(l);
// copy to a container of types ConvertibleTo
same_as<std::vector<long>> auto d = ranges::to<std::vector<long>>(l);


//Supports converting associative container to sequence containers
same_as<std::vector<std::pair<const int, int>>> auto f = ranges::to<vector>(m);

//Supports converting sequence containers to associative ones
same_as<std::map<int, int>> auto g = ranges::to<map>(f);

//Pipe syntaxe
same_as<std::vector<int>> auto g = l | ranges::view::take(42) | ranges::to<std::vector>();

//Pipe syntax with allocator
auto h = l | ranges::view::take(42) | ranges::to<std::vector>(alloc);

//The pipe syntax also support specifying the type and conversions
auto i = l | ranges::view::take(42) | ranges::to<std::vector<long>>();

// Nested ranges
std::list<std::forward_list<int>> lst = {{0, 1, 2, 3}, {4, 5, 6, 7}};
auto vec1 = ranges::to<std::vector<std::vector<int>>>(lst);
auto vec2 = ranges::to<std::vector<std::deque<double>>>(lst);
```

## Tony tables

| Before | After |
|---|---|
| `std::list<int> lst = /*...*/;`<br>`std::vector<int> vec`<br>`  {std::begin(lst), std::end(lst)};` | `std::vector<int> vec = lst \| ranges::to<std::vector>();` |
| `auto view = ranges::iota(42);`<br>`vector <`<br>`  iter_value_t<`<br>`  iterator_t<decltype(view)>`<br>`  >`<br>`> vec;`<br>`if constexpr(SizedRanged<decltype(view)>) {`<br>`  vec.reserve(ranges::size(view)));`<br>`}`<br>`ranges::copy(view, std::back_inserter(vec));` | `auto vec = ranges::iota(42)`<br>`  \| ranges::to<std::vector>();` |
| `std::map<int, widget> map = get_widgets_map();`<br>`std::vector<`<br>`  typename decltype(map)::value_type`<br>`> vec;`<br>`vec.reserve(map.size());`<br>`ranges::move(map, std::back_inserter(vec));` | `auto vec = get_widgets_map()`<br>`  \| ranges::to<vector>();` |

## Motivation

Most containers of the standard library provide a constructors taking a pair of iterators.

```
std::list<int> lst;
std::vector<int> vec{std::begin(lst), std::end(lst)};
//equivalent too
std::vector<int> vec;
std::copy(std::begin(lst), std::end(lst), std::back_inserter(vec));
```

While, this feature is very useful, as converting from one container type to another is a frequent use-case, it can be greatly improved by taking full advantage of the notions and tools offered by ranges.

Indeed, given all containers are ranges (ie: an iterator-sentinel pair) the above example can be rewritten, without semantic as:

```
std::list<int> lst;
```

```
std::vector<int> vec = lst | ranges::to<std::vector>();
```

The above example is a common pattern as it is frequently preferable to copy the content of a `std::list` to a `std::vector` before feeding it an algorithm and then copying it back to a `std::vector`.

As all containers and views are ranges, it is logical they can themselves be easily built out of ranges.

### View Materialization

The main motivation for this proposal is what is colloquially called *view materialization*. A view can generate its elements lazily (upon increment or decrement), such as the value at a given position of the sequence iterated over only exist transiently in memory if an iterator is pointing to that position. (Note: while all lazy ranges are views, not all views are lazy).

*View materialization* consists in committing all the elements of such view in memory by putting them into a container.

The following code iterates over the numbers 0 to 1023 but only one number actually exists in memory at any given time.

```
std::iota_view v{0, 1024};
for (auto i : v) {
    std::cout << i << ' ';
}
```

While this offers great performance and reduced memory footprint, it is often necessary to put the result of the transformation operated by the view into memory. The facilities provided by [**?**] allow to do that in the following way:

```
std::iota_view v{0, 1024};
std::vector<int> materialized;
std::copy(v, std::back_inserter(materialized));
```

This proposal allows rewriting the above snippet as:

```
auto materialized = std::iota_view{0, 1024} | std::ranges::to<std::vector>();
```

Perhaps the most important aspect of view materialization is that it allows simple code such as:

```
namespace std {
    split_view<std::string_view> split(std:std::string_view);
}
auto res = std::split("Splitting strings made easy")
           | std::ranges::to<std::vector>();
```

Indeed, a function such as `split` is notoriously hard to standardize ([**?**], [**?**]), because without lazy views and `std::string_view`, it has to allocate or expose an expert-friendly interface. The view materialization pattern further let the *caller* choose the best container and allocation strategy for their use case (or to never materialize the view should it not be necessary). And while it would not make sense for a standard-library function to split a string into a vector it would allocate, it's totally reasonable for most applications to do so.

This paper does not propose to standardize such `split` function - a `split_view` exist in [**?**], however, view materialization is something the SG-16 working group is interested in. Indeed, they have considered APIs that could rely heavily on this idiom, as it has proven a natural way to handle the numerous ways to iterate over Unicode text. Similar ideas have been presented in [**?**].

```
auto sentences =
    text(blob)
    normalize<text::nfc> |
    graphemes_view |
    split<sentences> | ranges::to<std::vector<std::u8string>>();
```

## Design

Conceptually, `to` is a function template with multiple overloads:

- One that is templated on a container type and converts a range to that type using the most efficient method depending on the type of that container.
- One that accepts a container as template parameter and deduced the value type of that container using CTAD
- One that offers a pipe adaptor object over both these overloads

Care was taken to support move only iterators, ranges of ranges, non const views, associative containers (in either direction)

### Should ranges::to be able to call `reserve`/`assigned` ?

For performance reasons, `ranges::to` should be able to reserve memory in the container before assigning the range whenever possible. This cannot be done in the constructor of individual ranges because a range meeting the requirements of `SizedRange` doesn't imply that the distance between each of the iterator forming the range can be computed efficiently, if at all.

Both cmcstl2 and ranges-v3 determine whether or not a container can be reserved and assigned by means of a `ReserveAndAssignable` concept that is not part of the user facing API, similar to the following

```
template <typename T, typename I >
concept ReserveAndAssignable =
```

```
        InputIterator<I> && requires (C &c, C const &cc, range_size_t<C> s, I i) {
                c.reserve(s);
                cc.capacity();
                { cc.capacity() } -> range_size_t<C>;
                c.assign(i, i);
};
```

LWG made the case that there should be a customization point for this behavior and that the set of requirements should be specified.

- Do we want implementation to be able to reserve the container memory before assigning the range (this was always the intent) ? Do we want to allow it for arbitrary (non standard) containers ?
- Do we want to document the set of requirements of reservable container as a Named requirements, exposition only concept, concept in the std namespace ?
- Do we want a customization point (similar to `disable_sized_range`)? (The authors believe the answer to that should be no - the risk of ambiguity is extremely low, we don't have an opt out for Container - in fact we don't have a Container concept at all), and I don't think that all concepts should have an opt-out.

For the purpose of `ranges::to` we further require that the container be constructible from the extra arguments passed to `ranges::to`.

A conservative approach would be to leave the design unchanged for now (implementers can do reserve + assign for standard types only), and revisit the question for non-standard containers at a later date.

The wording in this revision of this paper reflects that conservative approach.

## Alternative designs

While we believe the range constructor based approach is the cleanest way to solve this problem, LEWG was interested in an alternative design based on free functions

### Range constructors

The original version of that paper proposed to add range constructors to all constructors. This proved to be unworkable because of std::initializer_list:

```
std::list<int> foo = ....;
std::vector a{foo}; //constructs a std:vector<std::list<int>>
std::vector b(foo); //would construct a std::vector<int>
```

### Tagged range constructors

To solve the problem described above, it is possible to add a tag

```
std::vector<int> foo = ....;
std::vector a{std::from_range, foo}; //constructs a std:vector<int>
```

This will be explored in a separate paper by Tristan Brindle. However this approach does not replace ranges::to, which has the advantages of being a pipe adaptor and works with non-standard containers which do not support such tag constructors.

`ranges::to` can take advantages of these ranges constructors when available. The proposed wording in this paper body assumes that tagged constructors will be adopted in addition to the current paper.

## Existing practices

### Range V3

This proposal is based on the `to` (previously (`to_`) function offered by ranges v3.

```
auto vec = view::ints
| view::transform([](int i) {
        return i + 42;
})
| view::take(10)
| to<std::vector>;
```

### Abseil

Abseil offers converting constructors with each of their view. As per their documentation:

> One of the more useful features of the StrSplit() API is its ability to adapt its result set to the desired return type. StrSplit() returned collections may contain std::string, absl::string_view, or any object that can be explicitly created from an absl::string_view. This pattern works for all standard STL containers including std::vector, std::list, std::deque, std::set, std::multiset, std::map, and std::multimap, and even std::pair, which is not actually a container.

Because they can not modify existing containers, view materialization in Abseil is done by the mean of a conversion operator:

```
template<Container C>
operator C();
```

However, because it stands to reason to expect that there are many more views than containers and because conversions between containers are also useful, it is a more general solution to provide a solution that is not coupled with each individual view.

### Previous work

[**?**] explores similar solutions and was discussed by LEWG long before the Ranges TS.

## Proposed wording

Wording is relative to [**?**].

Add to the synopsis in [ranges.syn]:

```
namespace std::ranges {

        template <typename C, input_range R, typename... Args>
    requires (!view<C>)
        constexpr C to(R && r, Args&&...);

        template <template <typename...> typename C, input_range R, typename... Args>
        constexpr auto to(R && r, Args&&...) -> see below;

}
```

In [range.utility] Insert after section [range.dangling]

### 🚧 Container conversions [range.utility.container]

The container conversions functions provide functions that efficiently convert ranges to containers. The container is constructed from the begin(range)/end(range)iterators pair.

```
template <typename C, input_range R, typename... Args>
requires (!view<C>)
constexpr C to(R && r, Args&&... args);
```

Construct a container `C` with the elements of `r` in the following manner:

- If `std::constructible_from<C, R, Args...>` is true, equivalent to `C(std::forward<R>(r), std::forward<Args>(args)...)`.
- Otherwise, if `std::constructible_from<C, from_range_t, R, Args...>` is `true`, equivalent to `C(from_range, std::forward<R>(r), std::forward<Args>(args)...)`.
- Otherwise, if `std::constructible_from<C, Args...>` is `true` and `std::indirectly_copyable<ranges::range_iterator_t<R>, ranges::range_iterator_t<C>>` is `true`, equivalent to:

```
C c(std::forward<Args...>(args)...);
ranges::copy(std::forward<R>(r), std::inserter(c, std::end(c)));
```

- Otherwise, if:
  - `ranges::input_range<ranges::range_value_t<C>>` is `true` and
  - `ranges::input_range<ranges::range_value_t<R>>` is `true` and
  - `ranges::view<ranges::range_value_t<C>>` is `false` and

- `std::indirectly_copyable<ranges::range_iterator_t<ranges::range_value_t<R>>, ranges::range_iterator_t<ranges::range_value_t<C>>>` is `true`

equivalent to:

```
C c(std::forward<Args...>(args)...);
auto v = r | transform ([](auto && elem) {
    return to<range_value_t<C>>(elem);
});
ranges::copy(v, std::inserter(c, std::end(c)));
```

- Otherwise `ranges::to<C>(r, args)` is ill-formed.

```
template <template <typename...> typename C, input_range R, typename... Args>
constexpr auto to(R && r, Args&&... args) -> ContainerType;
```

Equivalent to `ranges::to<`*ContainerType*`>(std::forward<R>(r), std::forward<Args>(args)...)` where *ContainerType* is determined using CTAD as follow:

Let `ITER` be a type meeting the requirements of `Cpp17InputIterator` such that

- `same_as<`*ITER*`::iterator_category, input_iterator_tag>` is `true`,
- `same_as<`*ITER*`::value_type, ranges::range_value_t<R>>` is `true`,
- `same_as<`*ITER*`::difference_type, ranges::range_difference_t<R>>` is `true`,
- `same_as<`*ITER*`::pointer, ranges::range_value_t<R>*>` is `true`,
- `same_as<`*ITER*`::reference, ranges::range_reference_t<R>>` is `true`.

Then, `ContainerType` is:

- `decltype(C(std::declval<R>(), std::declval<Args>()...))` if that is a valid expression,
- Otherwise, `decltype(C(from_range, std::declval<R>(), std::declval<Args>()...))` if that is a valid expression,
- Otherwise, `decltype(C(std::declval<`*ITER*`>(), std::declval<`*ITER*`>(), std::declval<Args>()...))` if that is a valid expression,
- Otherwise `ranges::to<C>(r, args)` is ill-formed.

### 🔲 `ranges::to` adaptors [range.utility.container.adapters]

In addition to the functions described above, `ranges::to` also defines a closure object that accepts a `viewable_range` argument and returns a *Container* such that the expressions `r | ranges::to<Container>(args...)` and `ranges::to<Container>(r, args...)` have equivalent semantics. [*Note:* Container denotes either a class or a class template —*end note*].

The bitwise OR operator is overloaded for the purpose of chaining `ranges::to` to the end of an adaptor chain pipeline.

[ *Example:*

```
list<int> ints{0,1,2,3,4,5};

auto v1 = ints | ranges::to<vector>();
auto v2 = ints | ranges::to<vector<int>>();
auto v3 = ranges::to<vector>(ints);
auto v4 = ranges::to<vector<int>>(ints);

assert(v1 == v2 && v2 == v3 && v3 == v4);
```

*— end example* ]

## Implementation Experience

Implementations of this proposal are available on in the 1.0 branch of [**?**] and in [**?**]. Another implementation reflecting exactly this proposal is available on Github [**?**].

## Related Paper and future work

- P1989R0 [**?**] adds range and iterator constructor to `string_view`
- [**?**] adds iterator constructors to `stack` and `queue`
- [**?**] Provide facilities to implementing `span` constructors more easily.

Future work is needed to allow constructing `std::array` from *tiny-ranges*.

## Acknowledgements

We would like to thank the people who gave feedback on this paper, notably Christopher Di Bella, Arthur O'Dwyer, Barry Revzin and Tristan Brindle.

## References

[cmcstl2] https://github.com/CaseyCarter/cmcstl2/blob/a7a714a9159b08adeb00a193e77b782846b3b20e/include/stl2/detail/to.hpp

[RangeV3] Eric Niebler https://github.com/ericniebler/range-v3/blob/v1.0-beta/include/range/v3/to_container.hpp

[rangesnext] Corentin Jabot https://github.com/cor3ntin/rangesnext/blob/master/include/cor3ntin/rangesnext/to.hpp

[CTAD Ranges] Eric Niebler https://github.com/ericniebler/range-v3/blob/d284e9c84ff69bb416d9d94d029729dfb38c3364/include/range/v3/range/conversion.hpp#L140-L152

[P1391] Corentin Jabot *Range constructor for std::string_view* https://wg21.link/P1391

[P1394] Casey Carter, Corentin Jabot *Range constructor for std::span* https://wg21.link/P1394

[P1425] Corentin Jabot *Iterators pair constructors for stack and queue* https://wg21.link/P1425

[P1419] Casey Carter, Corentin Jabot *A SFINAE-friendly trait to determine the extent of statically sized containers* https://wg21.link/P1419

[P0896R3] Eric Niebler, Casey Carter, Christopher Di Bella *The One Range Ts Proposal* https://wg21.link/P0896

[P1004] Louis Dionne *Making std::vector constexpr* https://wg21.link/P1004

[P1004] Tom Honermann *Text_view: A C++ concepts and range based character encoding and code point enumeration library* https://wg21.link/P0244

[P0540] Laurent Navarro *A Proposal to Add split/join of string/string_view to the Standard Library* https://wg21.link/P0540

[N3593] Greg Miller *std::split(): An algorithm for splitting strings* https://wg21.link/N3593

[P1035] Christopher Di Bella *Input range adaptors* https://wg21.link/P1035

[Abseil] https://abseil.io/docs/cpp/guides/strings

[N3686] Jeffrey Yasskin *[Ranges] Traversable arguments for container constructors and methods* https://wg21.link/n3686

[P1072R1] Chris Kennelly, Mark Zeren *Vector as allocation transfer device* https://wg21.link/P1072

[1] [P0504R0] Jonathan Wakely *Revisiting in-place tag types for any/optional/variant* https://wg21.link/P0504R0