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

Document #:	D1206R2
Date:	2019-07-11
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 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 ([P1391] and [P1394] 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<std::vector<int>> auto a = ranges::to<std::vector<int>>(l);
//Specify an allocator
Same<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<std::vector<int>> auto c = ranges::to<std::vector>(l);
// copy to a container of types ConvertibleTo
Same<std::vector<long>> auto d = ranges::to<std::vector<long>>(l);


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

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

//Pipe syntaxe
Same<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>>();

//Pathenthesis are optional for template
Same<std::vector<int>> auto j = l | ranges::view::take(42) | ranges::to<std::vector>;
//and types
auto k = l | ranges::view::take(42) | ranges::to<std::vector<long>>;
```

Tony tables

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td>

```cpp
std::list<int> lst = /*...*/;
std::vector<int> vec
        {std::begin(lst), std::end(lst)};
```

</td><td>

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

</td></tr>
<tr><td>

```cpp
auto view = ranges::iota(42);
vector <
  iter_value_t<
        iterator_t<decltype(view)>
  >
> vec;
if constexpr(SizedRanged<decltype(view)>) {
  vec.reserve(ranges::size(view)));
}
ranges::copy(view, std::back_inserter(vec));
```

</td><td>

```cpp
auto vec = ranges::iota(42)
        | ranges::to<std::vector>;
```

</td></tr>
<tr><td>

```cpp
std::map<int, widget> map = get_widgets_map();
std::vector<
  typename decltype(map)::value_type
> vec;
vec.reserve(map.size());
ranges::move(map, std::back_inserter(vec));
```

</td><td>

```cpp
auto vec = get_widgets_map()
          | ranges::to<vector>;
```

</td></tr>
</table>

Motivation

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

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

```cpp
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 [P0896R3] 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 ([P0540], [N3593]), 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 [P0896R3], 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 [P1004].

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

Constructing views from ranges

Constructing standard views (`string_view` and `span`) from ranges is addressed in separate papers as the design space and the requirements are different:

- `string_view` : [P1391]
- `span` : [P1394]
- Work is being done to allow Ranges's iterators to be move only

As views are not containers, they are not constructible from `ranges::to`

Alternative designs

While we believe the range constructor based approach is the cleanest way to solve this problem, LEWG was interested in 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::vector<int> foo = ....;
std::vector a{foo}; //constructs a std:vector<std::vector<int>>
std::vector b(foo); //would construct a std::vector<int>
```

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 offer 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

[N3686] explores similar solutions and was discussed by LEWG long before the Ranges TS.

Proposed wording

```
namespace std::ranges {

        // exposition only
        template<typename Rng, typename Cont, typename Args...>
        concept convertible-to-container =
        Range<Cont>
        && (!View<Cont>)
        && MoveConstructible<Cont>
        && ConvertibleTo<range_value_t<Rng>, range_value_t<Cont>>
        && Constructible<Cont, range_common_iterator_t<Rng>,
```

```
        range_common_iterator_t<Rng>, Args...>;

    template <typename C, InputRange R, typename... Args>
    requires convertible-to-container<Rng, C>
    constexpr auto to(const R & r, Args&&...) -> C;

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

    // exposition only
    using common-iterator-type = common_iterator<iterator_t<R>, sentinel_t<R>>;


}
```

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 rvalue overloads of the function ranges::to return an implementation defined object that can be passed to the pipe operator, and provide an implementation defined way of creating a container of the appropriate type. The container is constructed from the begin(range)/end(range)iterators pair. [*Note:* Implementations are encouraged to reserve the container's memory when the range is a `SizedRange` — *end note*]

```
template <typename C, InputRange R, typename... Args>
convertible-to-container<Rng, C, Args...>
constexpr auto to(const R & r, Args&&... args) -> C;
```

Returns: `C{common-iterator-type(ranges::begin(r)), common-iterator-type(ranges::end(r)), std::forward<Args>(args)...}`

```
template <template <typename...> typename C,  InputRange R, typename... Args>
constexpr auto to(const R & r, Args&&... args) -> Ret (see below);
```

The return type `Ret` is determined in the following way:

- The type of the expression `C{range_common_iterator_t<R>{}, range_common_iterator_t<R>{}}, args...}` if it is valid
- Otherwise `C<range_value_t<R>, Args...>`

  Mandates: `convertible-to-container<Rng, C, Args...>` is true

  Returns: `Ret{common-iterator-type(ranges::begin(r)), common-iterator-type(ranges::end(r)), std::forward<Args>(args)...}`

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

In additions to the functions described above, `ranges::to` also defines a closure object that accepts a `ViewableRange` 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*] .

In the absence of arguments, `r | ranges::to<Container>` denotes a valid expression equivalent to `r | ranges::to<Container>()`.

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 | to<vector>();
auto v2 = ints | to<vector>;
auto v3 = ints | to<vector<int>>();
auto v4 = ints | to<vector<int>>;
auto v5 = ints | to<vector>(ints);
auto v6 = ints | to<vector<int>>(ints);

assert(v1 == v2 && v2 == v3 && v3 == v4 && v4 == v5 && v5 == v6);
```

— *end example*]

Implementation Experience

Implementations of this proposal are available on in the 1.0 branch of [RangeV3] and in [cmcstl2].

To make sure the parentheses are optional (`v| ranges::to<vector>;`) our implementations use a default constructed tag which dispatch through a function pointer. However, this have no runtime cost and doesn't suffer from the sames issues LEWG had about `std::in_place_tag` because no actual indirection takes place. We believe being able to omit the parenthesis is necessary so `ranges::to` remains consistent with the syntax of views adaptors, and is otherwise a nice quality of life improvement in a facility which we expect to be used frequently.

An implementation strategy to deduce the concrete type of a container, including associative containers is to use CTAD, as shown in [CTAD Ranges].

This approach does not necessitate special casing to handle associative containers.

A more naive approach (instantiating the type of the container directly from the type of the range's value type) can be used as fallback for cases where no deduction guide was declared

Related Paper and future work

- [P1391] adds range and iterator constructor to `string_view`
- [P1394] adds range and iterator constructor to `span`
- [P1425] adds iterator constructors to `stack` and `queue`
- [P1419] 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

[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