Range constructors for standard containers and views

Document #:	P1206R0
Date:	2018-10-07
Project:	Programming Language C++
Audience:	LWG

Reply-to:	Corentin Jabot <corentin.jabot@gmail.com> Christopher Di Bella <cjdb.ns@gmail.com>

1 Abstract

Most standard containers and views can be constructed from an iterators-pair. This paper, complementing [P0896R3], proposes that all standard views, containers and string classes be constructible from a range.

2 Tony tables

<table>
<tr><th>Before</th><th>After</th></tr>
<tr><td><pre>std::list<int> lst = /*...*/;
std::vector<int> vec
        {std::begin(lst), std::end(lst)};</pre></td><td><pre>std::vector<int> vec{lst};</pre></td></tr>
<tr><td><pre>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));</pre></td><td><pre>std::vector vec = ranges::iota(42);</pre></td></tr>
<tr><td><pre>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));</pre></td><td><pre>std::map<int, widget> map = get_widgets_map();
std::vector vec{std::move(map)};
//vector<const int, widget></pre></td></tr>
<tr><td><pre>void foo(string_view);
vector<char8_t> vec = get_some_unicode();
foo(string_view{vec.data(), vec.size()});</pre></td><td><pre>void foo(string_view);
vector<char8_t> vec = get_some_unicode();
foo(vec);</pre></td></tr>
</table>

3 Non-goal

As explained in the "Design consideration", this proposal focuses on explicit construction and does not propose implicit container conversion.

4 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(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 of performance changes, as:

```
std::list<int> lst;
std::vector<int> vec{lst};
```

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 built out of ranges. Note that most containers and views already provide constructors for iterator-pairs, which themselves represent a range. They also provide copy and move constructors for ranges of the same type (`std::vector` provide a copy constructor from another `std::vector`, etc). This proposal is a generalization of these existing features.

4.1 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::ranges::copy(v, std::back_inserter(materialized));
```

This proposal allows rewriting the above snippet as:

```
std::vector materialized = std::iota_view{0, 1024};
```

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);
}
std::vector<std::string> words = std::split("Splitting strings made easy");
```

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].

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

5 Design considerations

5.1 Ranges and sentinels

Iterators from the Ranges TS are not always compatible with iterators from the `std` namespace. Namely,

- They do not have the same set of requirements.
- `std`'s iterator do not support unbounded ranges and `Sentinel`
- Work is being done to allow Ranges's iterators to be move only

Therefore, in the general case, the iterator-pair constructor offered by standard containers cannot be used, but instead the `ranges::copy` should be used. Deferring to the design decisions of [P0896R3], we think it's better avoided not to have support for both type of iterator-pairs in the same overload set as to avoid breaking code in subtle ways.

Therefore, adding support for `ranges::`'s ranges seem the best solution to make `std::` containers constructible from objects meeting the requirements specified in the `ranges::` namespace.

Ranges are also a better, safer, stronger abstraction compared to iterator-pairs.

5.2 `explicit`

Because copy of containers is costly, the authors of this paper believe it is important that the range-based constructors for containers be `explicit`. However, there is a strong interest for this syntax to be supported:

```
container c = view | transform;
```

But, at the same time, the following pitfalls should be avoided:

```
auto map m = /*...*/;
vector a = m;   //implicit conversion map -> vector (O(n))
vector b = m;   //implicit conversion map -> vector (O(n))
```

—

```
void foo(const vector<type> &);
deque a =  /*...*/;
foo(a);  //implicit conversion deque -> vector (O(n))
foo(a);  //implicit conversion deque -> vector (O(n))
```

—

```
std::list<type> foo();
void bar(const vector<type> &);
bar(foo()); //implicit conversion vector -> list (O(n))
```

—

```
void foo(const vector<type> &);
auto view = zip(...);
foo(view); // View materialized once
foo(view); // View materialized twice
```

All the above example crystallize concerns over performances traps that would indubitably arise. Therefore we think it is to best follow the existing practice not to allow implicit copy construction from objects of different types.

But because expiring views can only be materialized once and can therefore not be considered a copy, we think it is reasonable that containers can be constructed *implicitly* from `rvalue-reference` views.

This compromise leads to some oddity because of the inability to distinguish between views over existing non-transformed data (`span`, `string_view`) from generators (`iota_view`, `transform_view`, etc).

Notably,

```
{
    vector<int> ints(42);
    deque dq = ints; // error, implicit conversion of container
}
{
    vector<int> ints(42);
    span view = ints; //Ok, no copy => implicit
    deque dq =  std::move(view);  //Ok, implicit construction from a view, but does a copy
}
```

This oddity, arise from an expressed desire to eat our cake and have it too, or more accurately, offer a convenient syntax for view materialization while avoiding implicit conversion of containers.

5.3 Movability

Beside being desirable to have different `explicit`-ness policies for containers and views, the content of `rvalue-reference Containers` can be moved-from, as if per `std::move_iterator` rather than copied. This is however generally undecidable for views which may not own the underlying data, and so views should only be copied-from.

Concerns were raised circa 2014 that constructors are proposed here would copy data from the view more often than necessary and that something akin to

```
view.to_container<vector>();
```

might be more suitable.

However, the authors think this question is worth reexamining given the evolution of the ranges TS over the past 4 years. Notably:

- It would be possible for **lazy** views to indicate that they can be moved from (through a tag).

- Alternatively, it might be worth considering whether deferencing an iterator over a lazy view should be recommended to return by value.
- The case can be made that for non-forward `InputIterator`, it is always reasonable to move-from the elements rather than copying them. This is explored at length in [P1207]
- In the general case, non-owning views don't have any knowledge of whether they can be moved-from.

5.4 Range constructor for views

Views (`span`, `basic_string_view`), can only be constructed from a `ContiguousRange` of the same type. Because they don't copy the data, they do not need to be explicit as constructing a view is cheap. On the other end, because they don't own the data, we must take care to only construct them from lvalue reference.

5.5 `constexpr`

Views (`std::span`, `std::basic_string_view`) constructors can be `constexpr` and so, they shall be. Other containers are currently not `constexpr`-constructible, but work is being down in this area. As more containers gain `constexpr` constructors, the range-based constructors as proposed here should be made `constexpr` too.

6 Existing practices

6.1 Abseil

View materialization is a technique notably adopted by the [Abseil] library. 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 accept ranges in container constructors than it is to make each view convertible to a container.

6.2 Range V3

The range-v3 offers a `to_<Container>` method which copy a `Range` into a Container c. It is interesting to note that, to the best understanding of the authors, this methods always perform a deep copy of each element, rather than a move, when it can.

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

6.3 Previous work

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

7 Future work

Whether `std::vector` can be converted to and from `std::string` in $\mathcal{O}(1)$ is an area of interest, notably for SG-16 - see [P1072R1]. Should such conversion exist, it should take precedence over the generic range-constructor proposed here.

8 Proposed wording

A more complete wording will ve provided in a subsequent revision

Change in [**basic.string**] **20.3.2**:

```
namespace std {
template<class charT, class traits = char_traits<charT>,
class Allocator = allocator<charT>>
class basic_string {
    public:

    [...]

    basic_string() noexcept(noexcept(Allocator())) : basic_string(Allocator()) { }
    explicit basic_string(const Allocator& a) noexcept;
    basic_string(const basic_string& str);
    basic_string(basic_string&& str) noexcept;
    basic_string(const basic_string& str, size_type pos, const Allocator& a = Allocator());
    basic_string(const basic_string& str, size_type pos, size_type n,
    const Allocator& a = Allocator());
    template<class T>
    basic_string(const T& t, size_type pos, size_type n, const Allocator& a = Allocator());
```

```
    template<class T>
    explicit basic_string(const T& t, const Allocator& a = Allocator());
    basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
    basic_string(const charT* s, const Allocator& a = Allocator());
    basic_string(size_type n, charT c, const Allocator& a = Allocator());
    template<class InputIterator>
    basic_string(InputIterator begin, InputIterator end, const Allocator& a = Allocator());
    basic_string(initializer_list<charT>, const Allocator& = Allocator());
    basic_string(const basic_string&, const Allocator&);
    basic_string(basic_string&&, const Allocator&);

    template<InputRange C>
    explicit basic_string(C&&, const Allocator& = Allocator());

    template<InputRange R>
    requires InputView<R>
    explicit(see-below)
    basic_string(R&&, const Allocator& = Allocator());
    ~basic_string();

    [...]
};

template<class InputIterator,
class Allocator = allocator<typename iterator_traits<InputIterator>::value_type>>
basic_string(InputIterator, InputIterator, Allocator = Allocator())
-> basic_string<typename iterator_traits<InputIterator>::value_type,
char_traits<typename iterator_traits<InputIterator>::value_type>,
Allocator>;

template<InputRange R,
class Allocator = allocator<iter_value_t <iterator_t< R>>>>
explicit() basic_string(R&& b, Allocator a = Allocator() )
 -> basic_string<
 iter_value_t<iterator_t<R>>,
 char_traits<iter_value_t<iterator_t<R>>>,
 Allocator
 >;

template<class charT,
class traits,
class Allocator = allocator<charT>>
explicit basic_string(basic_string_view<charT, traits>, const Allocator& = Allocator())
-> basic_string<charT, traits, Allocator>;

template<class charT,
class traits,
class Allocator = allocator<charT>>
basic_string(basic_string_view<charT, traits>,
typename see below::size_type, typename see below::size_type,
const Allocator& = Allocator())
```

```
-> basic_string<charT, traits, Allocator>;

}
```

Change in [**string.cons**] **20.3.2.2**:

Add after 23

```
template<InputRange C>
requires Constructible<charT, iter_value_t<iterator_t<C>>>
explicit basic_string(C&& r, const Allocator& = Allocator());
```

Effects: In a move constructor, constructs a string by moving from the elements of `r` in a way equivalent to

```
ranges::move(r, std::back_inserter(*this));
```

Otherwise, constructs a string from the values in the range [`ranges::begin(r)`, `ranges::end(r)`).

Complexity: Linear in `ranges::size(r)`.

—

```
template<InputRange R>
requires InputView<R>
requires Constructible<charT, iter_value_t<iterator_t<C>>>
explicit(see below)
basic_string(R&& r, const Allocator& = Allocator());
```

Effects: Constructs a string from the values in the range [`ranges::begin(r)`, `ranges::end(r)`)

Remarks: This constructor shall not participate in overload resolution unless

- `is_array<R>` is `false`.

Complexity: Linear in `ranges::size(r)`.

The expression inside explicit is equivalent to:

```
!is_rvalue_reference_v<V&&>
```

Add after 28

```
template<InputRange R, class Allocator = allocator<iter_value_t <iterator_t< R>>>>
explicit(see below) basic_string(R&& b, Allocator a = {})
        -> basic_string<
                iter_value_t<iterator_t<R>>,
                char_traits<iter_value_t<iterator_t<R>>>,
                Allocator>;
```

Remarks: Shall not participate in overload resolution if Allocator is a type that does not qualify as an allocator.

The expression inside explicit is equivalent to:

```
!View<R&&> && !is_rvalue_reference_v<V&&>
```

Change in [**string.view**] **20.4.2**:

```
template<class charT, class traits = char_traits<charT>>
class basic_string_view {
public:
        [...]

        // construction and assignment
        constexpr basic_string_view() noexcept;
        constexpr basic_string_view(const basic_string_view&) noexcept = default;
        constexpr basic_string_view& operator=(const basic_string_view&) noexcept = default;
        constexpr basic_string_view(const charT* str);
        constexpr basic_string_view(const charT* str, size_type len);

        template <ContiguousRange R>
        requires Same<iter_value_t<iterator_t<R>>, charT>
        constexpr basic_string_view(const R& r);

        [...]
};

template<ContiguousRange R>
basic_string_view(const R& b)
    -> basic_string_view<
        iter_value_t<iterator_t<R>>,
        char_traits<iter_value_t<iterator_t<R>>>
    >;
```

Change in [**string.view.cons**] **20.4.2.1**:

Add after 7

```
template <ContiguousRange R>
requires Same<iter_value_t<iterator_t<R>>, charT>
constexpr basic_string_view(const R& r);
```

Requires: r is a valid range. *Effects:* Constructs a `basic_string_view`, with the over `ContiguousRange` r.

Remarks: This constructor shall not participate in overload resolution unless

- `is_array<R>` is `false`.

8.1 Yet to be provided wording for

- vector
- deque

- list
- forward_list
- priority_queue
- map
- multimap
- set
- multiset
- unordered_map
- unordered_set
- unordered_multiset
- unordered_multimap
- span (notably, we wish to modify span to be constructed from a ContiguousRange rather than a container, for the sake of consistency.)

9 Acknowledgements

We would like to thank the people who gave feedback on this paper, notably Casey Carter, Arthur O'Dwyer, Barry Revzin and Tristan Brindle.
We would also further acknowledge that this paper can only exist because of the incredible body of work constituting the Ranges TS.

10 References

[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

[P1207] Corentin Jabot *Movability of Single-pass Iterators*
https://wg21.link/P1207

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