Range constructor for std::string_view

Document #: P1391R2
Date: 2019-06-17
Project: Programming Language C++
Audience: LEWG, LWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

1 Abstract

This paper proposes that `string_view` be constructible from any contiguous range of characters. The idea was extracted from P1206.

2 Tony tables

Before	After

`void foo(string_view);`
`vector<char8_t> vec = get_some_unicode();`
`foo(string_view{vec.data(), vec.size()});`
	`void foo(string_view);`
	`vector<char8_t> vec = get_some_unicode();`
	`foo(vec);`

3 Motivation

While P1206 gives a general motivation for range constructors, it's especially important for string_view because there exist in a lot of codebases string types that would benefit from being convertible to string_view `string_view`. For example, `llvm::StringRef`, `QByteArray`, `fbstring`, `boost::container::string` ...

Manipulating the content of a vector as a string is also useful.

Finally, this makes contiguous views operating on characters easier to use with `string_view`.

4 Design considerations

- instantiations of `basic_string` are specifically excluded because `std::basic_string` already provides a conversion operator and more importantly, strings with different char_traits should not be implicitly convertible
- Because `basic_string_view` doesn't mutate the underlying data, there is no reason to accept a range by something other than const lvalue reference.

- The construction is implicit because it is cheap and a contiguous range of character is the same platonic thing as a string_view.

5 Arrays and null terminated strings

During review by LWG, it was noticed that the proposed change introduces this arguably surprising behavior:

```
char const t[] = "text";
std::string_view s(t); // s.size() == 4;


std::span<char> tv(t);
std::string_view s(tv); // s.size() == 5;
```

This is not an ambiguity of the overload set but rather a consequences of both null-terminated terminated strings and array of characters being both sequence of characters with array of characters implicitly convertible to pointers.

To be consistent with C++17 and not introduce a behavior change, we make sure arrays of characters decay to const charT*. We think this proposed design is consistent with existing practices of having to be explicit about the size in the presence of embedded nulls as well as the general behavior of C functions, and does not introduce a new problem - how unfortunate that problem might be. It is also worth noting that while embedded nulls have a lot of known usages they are not the common case.

Finding a better solution to that problem is not possible at the level of this proposal and would require major breaking language changes.

6 Proposed wording

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 <typename R>
    constexpr basic_string_view(const R& r);

    template <typename It, typename End>
    constexpr basic_string_view(It begin, End end);

    [...]
};
template<class R>
basic_string_view(const R&)
 -> basic_string_view<remove_reference_t<iter_reference_t<ranges::iterator_t<R>>>>;
template<class It, class End>
basic_string_view(It, End) -> basic_string_view<remove_reference_t<iter_reference_t<It>>>;
```

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

Add after 7

```
template <typename R>
constexpr basic_string_view(const R& r);
```

Constraints:

- `R` models `ranges::ContiguousRange`,
- `ranges::SizedRange<const R>` is true,
- `Same<iter_value_t<iterator_t<const R>>, charT>` is true,
- `is_convertible_v<const R&, const charT*>` is `false`,
- `R` does not derive from a specialization of `std::basic_string`,
- `R` does not derive from a specialization of `std::basic_string_view`.

Effects: Constructs a `basic_string_view` over the `ContiguousRange` `r` by initializing `data_` with `ranges::data(r)` and `size_` with `ranges::size(r)`.

Throws: What and when `ranges::data(r)` and `ranges::size(r)` throw.

```
template <typename It, typename End>
constexpr basic_string_view(It begin, End end);
```

Constraints:

- `It` models `ContiguousIterator`,
- `End` models `SizedSentinel<It>`,
- `Same<iter_value_t<It>, charT>` is true,
- `It` does not derive from an instantiation of `std::basic_string::iterator` or `std::basic_string::const_iterator`,
- `It` does not derive from an instantiation of `std::basic_string_view::iterator`, `std::basic_string_view::const_iterator`,

- `It` and `End` are not of the same type or `End` is not convertible to a pointer of `charT`.

Expects: `[begin, end)` shall be a valid range.

Effects: Constructs a `basic_string_view` over the range [begin, end) by initializing `data_` with `data(begin)` and `size_` with `distance(begin, end)`.

Add a new section [string.view.deduction] to describe the following deduction guides:

```
template <class It, class End>
basic_string_view(It, End) -> basic_string_view<remove_reference_t<iter_reference_t<It>>>
```

Constraints:

- `It` models `ranges::ContiguousIterator`,
- `End` models `SizedSentinel<It>`.

```
template<class R>
basic_string_view(const R&)
-> basic_string_view<remove_reference_t<iter_reference_t<ranges::iterator_t<R>>>
```

Constraints: `R` models `ranges::ContiguousRange`.