

Aligning span and string_view

Abstract

This paper proposes extensions to span and string_view, unifying their syntax for common subsetting and shrinking operations. Additionally it proposes a subset of the aforementioned features for string as well in order to enable generic handling of it and string_view.

Tony Table

Before	Proposed
<pre>string_view v = ...; span<T> p = ...; string s = ...;</pre>	<pre>string_view v = ...; span<T> p = ...; string s = ...;</pre>
<pre>//removing leading elements v.remove_prefix(5); p = p.subspan(5); s.erase(s.begin(), s.begin() + 5); //removing trailing elements v.remove_suffix(3); p = p.subspan(0, p.size() - 3); s.erase(s.end() - 3, s.end());</pre>	<pre>//removing leading elements v.remove_prefix(5); p.remove_prefix(5); //NOT proposed: s.remove_prefix(5); //removing trailing elements v.remove_suffix(3); p.remove_suffix(3); //NOT proposed: s.remove_suffix(3);</pre>
<pre>//getting leading elements auto l4v = v.subview(0, 4); auto l4p = p.first(4); auto l4s = s.subview(0, 4); //getting trailing elements auto t2v = v.subview(v.size() - 2); auto t2p = p.last(2); auto t2s = s.subview(s.size() - 2);</pre>	<pre>//getting leading elements auto l4v = v.first(4); auto l4p = p.first(4); auto l4s = s.first(4); //getting trailing elements auto t2v = v.last(2); auto t2p = p.last(2); auto t2s = s.last(2);</pre>

Revisions

R0: Initial version

Motivation

Both span and string_view model non-owning references into contiguous memory. Whilst there is good reason for both of them to exist, there is little to no reason for them not providing the same APIs for shrinking the referenced memory region. This paper primarily aims to fix this inconsistency.

¹ RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at

Design Space

span-specific functionality

span provides both APIs for compile-time as well as runtime subsetting. The former is out of context for this paper as string_view is always of dynamic extent. The runtime subsetting API is defined as follows:

```
template<typename T, size_type E = dynamic_extent>
struct span {
...
    //runtime subsetting:
    constexpr span subspan(size_type pos = 0, size_type n = dynamic_extent) const;
    constexpr span first(size_type count) const;
    constexpr span last(size_type count) const;
...
};
```

Whilst subspan is conceptually already provided in the form of string_view::subview, first and last are missing for no apparent reason. We propose to add versions of them to string_view (as well as string to enable generic handling), changing their return types to string_view.

string_view-specific functionality

In addition to the aforementioned subview, string_view provides an easy to use API for shrinking the referenced memory area:

```
template<typename charT, typename traits = char_traits<charT>>
struct basic_string_view {
...
    //in place shrinking:
    constexpr void remove_prefix(size_type n);
    constexpr void remove_suffix(size_type n);
...
};
```

Whilst these functions are not applicable for fixed-size spans, they are perfectly useable for dynamically sized ones. We therefore propose to adopt them verbatim to the latter.

Contrary to our proposal to add first and last to string on the basis of consistency, we refrain from doing the same for remove_prefix and remove_suffix as they represent a fundamentally different operation for an owning container.

Summary

To summarise the proposed subsetting and shrinking APIs for the respective types:

```
template<typename charT, typename traits = char_traits<charT>>
struct basic_string_view {
...
    //runtime subsetting:
    constexpr basic_string_view substr(size_type pos = 0, size_type n = npos) const;
    constexpr basic_string_view subview(size_type pos = 0, size_type n = npos) const;
    constexpr basic_string_view first(size_type count) const;
    constexpr basic_string_view last(size_type count) const;
...
    //in place shrinking:
    constexpr void remove_prefix(size_type n);
    constexpr void remove_suffix(size_type n);
...
};

template<typename charT, typename traits = char_traits<charT>, typename Allocator = allocator<charT>>
struct basic_string {
...
    //runtime subsetting:
    constexpr basic_string substr(size_type pos = 0, size_type n = npos) const;
    constexpr basic_string_view<charT, traits> subview(size_type pos = 0, size_type n = npos) const;
    constexpr basic_string_view<charT, traits> first(size_type count) const;
    constexpr basic_string_view<charT, traits> last(size_type count) const;
...
};
```

```

template<typename T, size_type E = dynamic_extent>
struct span {
...
    //compile-time subsetting:
    template<size_t Offset, size_t Count = dynamic_extent>
    constexpr span<T, /*see below*/> subspan() const;
    template<size_t Count>
    constexpr span<T, Count> first() const;
    template<size_t Count>
    constexpr span<T, Count> last() const;

    //runtime subsetting:
    constexpr span subspan(size_type pos = 0, size_type n = dynamic_extent) const;
    constexpr span first(size_type count) const;
    constexpr span last(size_type count) const;

    //in place shrinking:
    constexpr void remove_prefix(size_type n) requires(E == dynamic_extent); //this proposal
    constexpr void remove_suffix(size_type n) requires(E == dynamic_extent); //this proposal
...
};

```

Impact on the Standard

This proposal is a pure library addition. Existing standard library classes are modified in a non-ABI-breaking way.

Proposed Wording

Wording is relative to [N5008], with the plenary-approved changes from P3044. Additions are presented like [this](#), removals like ~~this~~ and drafting notes like [this](#).

[version.syn]

```

#define __cpp_lib_string_first_last 202606YYYYMMML //also in <string>, <string_view>
[DRAFTING NOTE: Applies to string and string_view, as it is expected these changes will be implemented in unison.]

#define __cpp_lib_span 202311YYYYMMML //freestanding, also in <span>

[DRAFTING NOTE: Adjust the placeholder values as needed to denote the proposal's date of adoption.]

```

[string.view.template]

??? General <pre> namespace std { template<class charT, class traits = char_traits<charT>> class basic_string_view { ... // [string.view.ops], string operations ... constexpr basic_string_view subview(size_type pos = 0, size_type n = npos) const; constexpr basic_string_view first(size_type count) const; constexpr basic_string_view last(size_type count) const; constexpr int compare(basic_string_view s) const noexcept; ... }; } </pre> ... String operations <pre> constexpr basic_string_view substr(size_type pos = 0, size_type n = npos) const; constexpr basic_string_view subview(size_type pos = 0, size_type n = npos) const; </pre> 10 Throws: out_of_range if pos > size(). constexpr basic_string_view first(size_type count) const; 11 Hardened preconditions: count <= size() is true. 12 Effects: Equivalent to: return {data(), count}; constexpr basic_string_view last(size_type count) const;	[string.view.template.general] // freestanding-deleted [string.view.ops]
---	---

```

13     Hardened preconditions: count <= size() is true.
14     Effects: Equivalent to: return {data() + (size() - count), count};
constexpr int compare(basic_string_view str) const noexcept;
145     Let rlen be the smaller of size() and str.size().
...

```

[basic.string]

????? General

[basic.string.general]

...

- 3 In all cases, [data(), data() + size()] is a valid range, data() + size() points at an object with value charT() (a “null terminator”), and size() <= capacity is true.

```

namespace std {
    template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
    class basic_string {
    ...
        // [string.ops], string operations
    ...
        constexpr basic_string_view<charT, traits> subview(size_type pos = 0, size_type n = npos) const;
        constexpr basic_string_view<charT, traits> first(size_type count) const;
        constexpr basic_string_view<charT, traits> last(size_type count) const;

        template<class T>
        constexpr int compare(const T& t) const noexcept(see below);
    ...
    };
}

```

...

????? basic_string::substr

[string.substr]

...

```
constexpr basic_string_view<charT, traits> subview(size_type pos = 0, size_type n = npos) const;
```

- 3 *Effects: Equivalent to: return basic_string_view<charT, traits>(*this).subview(pos, n);*

```
constexpr basic_string_view<charT, traits> first(size_type count) const;
```

- 4 *Effects: Equivalent to: return basic_string_view<charT, traits>(*this).first(count);*

```
constexpr basic_string_view<charT, traits> last(size_type count) const;
```

- 5 *Effects: Equivalent to: return basic_string_view<charT, traits>(*this).last(count);*

????? basic_string::compare

[string.compare]

[views.span]

????? Overview

[span.overview]

...

- 2 All member functions of span have constant time complexity.

```

namespace std {
    template<class ElementType, size_t Extent = dynamic_extent>
    class span {
    ...
        // [span.sub], subviews
    ...
        constexpr span<element_type, dynamic_extent> subspan(
            size_type offset, size_type count = dynamic_extent) const;

        // [span.mod], modifiers
        constexpr void remove_prefix(size_type n);
        constexpr void remove_suffix(size_type n);

        // [span.obs], observers
    ...
    };
}

```

...

[DRAFTING NOTE: Add a new subsection [span.mod] between [span.sub] and [span.obs] with the following content:]

```
constexpr void remove_prefix(size_type n);
1      Constraints: Extent == dynamic_extent is true.
2      Hardened preconditions: n <= size() is true.
3      Effects: Equivalent to: data_ += n; size_ -= n;
constexpr void remove_suffix(size_type n);
4      Constraints: Extent == dynamic_extent is true.
5      Hardened preconditions: n <= size() is true.
6      Effects: Equivalent to: size_ -= n;
```

Acknowledgements

Thanks to RISC Software GmbH for supporting this work.