

Document number: P3044R2  
Date: 2025-06-19  
Project: Programming Language C++  
Audience: LWG  
Reply-to: Michael Florian Hava<sup>1</sup> <[mfh.cpp@gmail.com](mailto:mfh.cpp@gmail.com)>

# sub-string\_view from string

## Abstract

This paper proposes a way to retrieve a sub-string\_view from a string directly.

## Tony Table

Before	Proposed
<pre>string s{"Hello cruel world!"}; auto sub = string_view{s}.substr(6); //sub == "cruel world!" auto subsub = sub.substr(0, 5); //subsub == "cruel"</pre>	<pre>string s{"Hello cruel world!"}; auto sub = s.subview(6); //sub == "cruel world!" auto subsub = sub.subview(0, 5); //subsub == "cruel"</pre>

## Revisions

**R0:** Initial version

**R1:** Changes after LEWG Mailing List Review in May 2024:

- Added discussion on naming and reference qualification

**R2:** Rebased on N5008

- Marked string\_view::subview as *freestanding-deleted*

## Motivation

Whilst the concept of a non-owning reference into a string has been established decades ago<sup>2</sup>, the idea only got introduced into the standard library with the adoption of string\_view into C++17. The integration of which into string can only be classified as being limited to the role of a sink-only type - several member functions support inputs in the form of string\_view, yet none return a string\_view.

Given the "reduced" interface of string<sup>3</sup>, there is exactly one member function that would most likely return a string\_view if we were to design this part of the standard library just now: substr(...) const & - from the authors experience, said member function is never invoked in a context requiring an immediate copy.

---

<sup>1</sup> RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, [michael.hava@risc-software.at](mailto:michael.hava@risc-software.at)

<sup>2</sup> e.g. <https://help.perforce.com/sourcepro/11/html/toolsref/rwcssubstring.html> dates back to the 1990s.

<sup>3</sup> Compared to "kitchen-sink" designs in other environments.

## Design Space

As changing the return type of `substr` is not possible for obvious compatibility reasons, we instead propose a new member function `subview` as accessor to sub-views of a string (following established naming practice like `span::subspan` and `string::substr`), replicating the interface and design of `substr` in all but return type and reference qualifications:

```
template<typename charT, typename traits = char_traits<charT>, typename Allocator = allocator<charT>>
struct basic_string {
...
    constexpr basic_string substr(size_type pos = 0, size_type n = npos) const &;
    constexpr basic_string substr(size_type pos = 0, size_type n = npos) &&;
    constexpr basic_string_view<charT, traits> subview(size_type pos = 0, size_type n = npos) const;
...
};
```

In order to improve generically handling both `string` and `string_view`, we further propose to add `string_view::subview` as an alternate spelling of `string_view::substr`:

```
template<typename charT, typename traits = char_traits<charT>>
struct basic_string_view {
...
    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;
...
};
```

## Naming Discussion

As `str.subview()` is valid, it has been suggested that a more appropriate name for this functionality would be `view`, pointing to `basic_stringbuf` and `basic_[io]stringstream` as existing naming practice. We push back against these suggestions as we are aiming at providing an efficient alternative to `substr` and said usage is purely incidental to mirroring the existing API.

If people feel strongly there should be a convention of providing a `view` member function as a way to convert a „stringy type“ to the respective `string_view`, this should be done with a separate paper enumerating all missing occurrences of said function - the authors are aware of at least one additional type to be included in said paper: `basic_format_string`.

## Reference Qualification

It has been suggested to keep lvalue-reference qualifiers for `subview` to prevent some immediately dangling constructs. We don't consider this appropriate as it inadvertently renders perfectly safe code ill-formed as well:

	w/o reference qualifiers	w/ reference qualifiers
<code>auto s = getString().subview(...);</code>	⚠ immediately dangling	✓ dangling prevented
<code>getStringView(getString().subview(...));</code>	✓ valid code, never dangling	✗ ill-formed

Whilst we acknowledge the dangers of dangling, we consider it something that shouldn't be tackled in an ad-hoc way conflating lifetime and value categories, but something that should be handled properly for all „reference types“.

## Future Extension: subspan From Contiguous Containers?

A related functionality to this paper is imaginable: Adding `subspan` to contiguous containers (`array`, `inplace_vector`, `string`, `string_view`, `vector`). This is potentially more contentious as it would add a dependency to `span`/`<span>` to all these currently independent classes/headers, whereas the proposed `subview` does not.

**Proposed Poll:** LEWG is interested in a paper on `subspan` for contiguous containers.

# Impact on the Standard

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

# Implementation Experience

The proposed design has been implemented at: <https://github.com/MFHava/STL/tree/P3044>.

# Proposed Wording

Wording is relative to [N5008]. Additions are presented like **this**, removals like ~~this~~ and drafting notes like **this**.

## [version.syn]

```
#define __cpp_lib_string_subview YYYYMMML //also in <string>, <string_view>
```

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

## [string.view]

??? Class template **basic\_string\_view**

[string.view.template]

???? General

[string.view.template.general]

```
namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_string_view {
    public:
        ...
        // [string.view.ops], string operations
        ...
        constexpr basic_string_view substr(size_type pos = 0,
                                          size_type n = npos) const;           // freestanding-deleted
        constexpr basic_string_view subview(size_type pos = 0,
                                          size_type n = npos) const;           // freestanding-deleted
        constexpr int compare(basic_string_view s) const noexcept;
    };
}
```

???? String operations

[string.view.ops]

...

```
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;
```

7 Let rlen be the smaller of n and size() - pos.

## [basic.string]

??? Class template **basic\_string**

[basic.string]

...

???? General

[basic.string.general]

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

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

????? **basic\_string::substr**

[string.substr]

```

...
constexpr basic_string substr(size_type pos = 0, size_type n = npos) &&
2     Effects: Equivalent to: return basic_string(std::move(*this), pos, n);
    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);
    ????.?? basic_string::compare [string.compare]

```

## Acknowledgements

Thanks to [RISC Software GmbH](#) for supporting this work. Thanks to Jeff Garland for bringing this issue to my attention. Thanks to Peter Kulczycki for proofreading revision 1 of this paper.