Document number: P3044R0
Date: 2023-01-16
Project: Programming Language C++
Audience: LEWG
Reply-to: Michael Florian Hava[1] <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

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

Revisions
R0: Initial version

Motivation
Whilst the concept of a non-owning reference into a string has been established decades ago[2], 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`[3], 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.

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:

---

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

[2] e.g. https://help.perforce.com/sourcepro/11/html/toolsref/rwcsubstring.html dates back to the 1990s.

[3] Compared to "kitchen-sink" designs in other environments.

```
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;
…
};
```

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 [N4964]. Additions are presented like this, removals like ~~this~~ and drafting notes like **this**.

[version.syn]

```
#define __cpp_lib_string_subview YYYYML //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;
    constexpr basic_string_view subview(size_type pos = 0, size_type n = npos) const;
    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.