std::cstring_view - a C compatible std::string_view adapter

Introduction

We propose standardizing a small variation of std::string_view, called std::cstring_view, that has a null-terminated class invariant.

The proposed std::cstring_view is simply a forwarding adapter around std::string_view. Operations are delegated to a private member std::string_view, with some minor modifications to the interface to maintain the null-terminated class invariant.

Most importantly, std::cstring_view (like std::string) has a .c_str() member function for compatibility with C.

Motivation

There are 15 million github hits for “.c_str()” - mostly calls to std::string’s .c_str() method.

This is because:

1. Most C++ programs use C libraries and APIs.
2. Most C libraries and APIs use null-terminated strings.

Note that std::string didn’t originally have a null-terminated class invariant, but one was added quickly due to demand for it (for C compatibility).

Lack of null-termination is a common reason given for not adopting std::string_view in contexts where C libraries are frequently used.
Most C APIs accept strings via a pointer to the first character of a character sequence that is null-terminated (eg const char*). std::string and string literals are null-terminated, and so can be converted inexpensively to such a pointer (via, respectively, .c_str() or by array-to-pointer standard conversion)

A variation of std::string_view that can be converted inexpensively to such a pointer would therefore be likewise useful. We therefore propose such a variation (that maintains a null-terminated class invariant)

Q1. Why not use std::string?

std::string is not a view. It is not non-owning. It is expensive to construct. Usually dynamic memory has to be allocated for it to own, and then the string data has to be copied into the memory. std::cstring_view is non-owning, like std::string_view, and therefore is cheap to construct.

Q2. Why not use std::string_view?

std::string_view doesn’t have a null-terminated class invariant, and therefore is not compatible with C APIs. std::cstring_view does have this invariant, and therefore is compatible with C APIs.

Q3. Why not convert the std::string_view to a std::string?

See “Why not use std::string?”

Q4. Why not use const std::string&?

const std::string& will not “bind” to string literals and other null-terminated string types, such as those passed from C APIs to C++ programs, third party string classes, or indeed any character array subsequence with null-termination. std::cstring_view does.
Q5. Why not use const std::string& but construct (implicitly or explicitly) a temporary std::string from the other string types for it to bind to?

See “Why not use std::string?”

Q6. Why not use const char*?

Several reasons:

1. const char* does not store the length of the string. std::cstring_view, like std::string_view, does. See “Storing the size” below for a study of the tradeoffs.

2. std::cstring_view, like std::string_view, supports the convenient C++ string functionality expected of a C++ string type. It supports comparison operators on its string content with other std::cstring_views and the other string types. std::cstring_view has a standard hash function. It is compatible as an element type with the standard containers (sets/maps, unordered sets/maps) and algorithms library. It has convenient member functions and free functions for a variety of tasks. const char* doesn’t.

3. std::cstring_view captures more of the intention of the programmer over const char*. Formally the type const char* doesn’t imply that the pointed to char be the start of a null-terminated string. To the compiler it could literally be a pointer to a single char that may not be modified. std::cstring_view has a constraint that it is pointing to a null-terminated string. This means the compiler could potentially check it in a constraint checking mode.

Storing the size

As the character sequence referred to by std::cstring_view is null terminated, it would be possible to imagine an alternative design where std::cstring_view only held a pointer, and unlike std::string_view, did not hold the size.

We list the tradeoffs here:

<table>
<thead>
<tr>
<th>Use Case</th>
<th>No Size</th>
<th>With Size (Proposed)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert from <code>std::string</code></td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Convert from string literal</td>
<td>compile-time</td>
<td>compile-time</td>
</tr>
<tr>
<td>Convert from <code>const char*</code></td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Convert from <code>string_view</code></td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Convert from <code>ptr,size</code></td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Convert to <code>const char*</code></td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Convert to <code>std::string</code></td>
<td>O(N) but slower</td>
<td>O(N) but faster</td>
</tr>
<tr>
<td>Convert to <code>string_view</code></td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Convert to <code>ptr,size</code></td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td><code>.size()</code></td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Reinterpret cast <code>const char*</code></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Reverse iterable / searchable</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><code>operator==</code></td>
<td>Slower</td>
<td>Faster</td>
</tr>
<tr>
<td>Security</td>
<td>Less</td>
<td>More</td>
</tr>
<tr>
<td>Object Size</td>
<td>1 word</td>
<td>2 words</td>
</tr>
</tbody>
</table>

After studying this list we feel that on the whole, storing the size is the better choice, hence it is what we propose.

A third alternative was considered whereby the size was calculated lazily (ie the mutable size field could hold “-1” initially and then filed out when it was first requested) but it was felt that the extra branch everywhere would counteract the advantage and having a mutable member would complicate the constexpr interface.

**Best Practice**

Our (non-normative) thoughts on how `std::cstring_view` would be applied are:

If you are designing a function that needs to accept a string type parameter and one of:
- The function may need to forward the parameter as an argument to a C function with a const char* parameter; or
- (recursively) The function may need to forward the parameter as an argument to a function with a std::cstring_view parameter; or
- The function otherwise needs null-termination of the string.

...then use std::cstring_view as the parameter type.

Likewise, if you are building a non-owning data structure of string data, and those will be used in the ways described above, use std::cstring_views. Or, if you know the input source is always null terminated, no point in discarding the invariant, use a std::cstring_view.

Design

We will describe the design in an imperative fashion.

1. Duplicate the basic_string_view template in the <string_view> header.
2. Rename duplicate to basic_cstring_view
3. Remove member function implementations and private members.
4. Add private member basic_string_view sv
5. Implement member functions by delegating to sv
6. Remove .remove_suffix modifier (violates invariant)
7. Remove .substr and replace with two overloads. Unary returns basic_cstring_view, Binary returns basic_string_view
8. Add .c_str() member
9. Add operator basic_string_view implicit conversion
10. Add to std::basic_string implicit conversion to std::basic_cstring_view
11. Add null-terminated invariant to constructors requirements
12. Add std::null_terminated tag type.
13. Replace basic_cstring_view(const CharT* s, size_type count) with
   basic_cstring_view(std::null_terminated_t, const CharT* s, size_type count) and
   basic_cstring_view(std::null_terminated_t, const basic_string_view&)

Specification

Class template basic_string [basic.string]
namespace std {
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
class basic_string {
    public:

    // [string.ops], string operations
    const charT* c_str() const noexcept;
    const charT* data() const noexcept;
    charT* data() noexcept;
    operator basic_string_view<charT, traits>() const noexcept;
    operator basic_cstring_view<charT, traits>() const noexcept;
    allocator_type get_allocator() const noexcept;

    ...
};

Header <string_view> synopsis [string.view.synop]

namespace std {
    // [string.view.template], class template basic_string_view
    template<class charT, class traits = char_traits<charT>>
class basic_string_view;

    // [string.view.ctemplate], class template basic_cstring_view
    template<class charT, class traits = char_traits<charT>>
class basic_cstring_view;

    struct null_terminated_t {} null_terminated;

    // [string.view.comparison], non-member comparison functions
    template<class charT, class traits>
    constexpr bool operator==(basic_string_view<charT, traits> x,
                              basic_string_view<charT, traits> y) noexcept;

    template<class charT, class traits>
    constexpr bool operator!=(basic_string_view<charT, traits> x,
                              basic_string_view<charT, traits> y) noexcept;

    template<class charT, class traits>
    constexpr bool operator<( basic_string_view<charT, traits> x,
                               basic_string_view<charT, traits> y) noexcept;
};
template<class charT, class traits>
  constexpr bool operator==(basic_string_view<charT, traits> x, basic_string_view<charT, traits> y) noexcept;

template<class charT, class traits>
  constexpr bool operator<=(basic_string_view<charT, traits> x, basic_string_view<charT, traits> y) noexcept;

template<class charT, class traits>
  constexpr bool operator>=(basic_string_view<charT, traits> x, basic_string_view<charT, traits> y) noexcept;

// see [string.view.comparison], sufficient additional overloads of comparison functions

// [string.view.io], inserters and extractors
template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, basic_string_view<charT, traits> str);

//basic_string_view typedef names
using string_view = basic_string_view<char>;
using u8string_view = basic_string_view<char8_t>;
using u16string_view = basic_string_view<char16_t>;
using u32string_view = basic_string_view<char32_t>;
using wstring_view = basic_string_view<wchar_t>;
using cstring_view = basic_cstring_view<char>;
using u8cstring_view = basic_cstring_view<char8_t>;
using u16cstring_view = basic_cstring_view<char16_t>;
using u32cstring_view = basic_cstring_view<char32_t>;
using wcstring_view = basic_cstring_view<wchar_t>;

// [string.view.hash], hash support
template<class T> struct hash;
template<> struct hash<string_view>;
template<> struct hash<u8string_view>;
template<> struct hash<u16string_view>;
template<> struct hash<u32string_view>;
template<> struct hash<wstring_view>;
template<> struct hash<cstring_view>;
template<> struct hash<u8cstring_view>;
template<> struct hash<u16cstring_view>;
template<> struct hash<u32cstring_view>;
template<> struct hash<wcstring_view>;

inline namespace literals {
inline namespace string_view_literals {
    // [string.view.literals], suffix for basic_string_view literals
    constexpr string_view operator"sv(const char* str, size_t len) noexcept;
    constexpr u8string_view operator"sv(const char8_t* str, size_t len) noexcept;
    constexpr u16string_view operator"sv(const char16_t* str, size_t len) noexcept;
    constexpr u32string_view operator"sv(const char32_t* str, size_t len) noexcept;
    constexpr wstring_view operator"sv(const wchar_t* str, size_t len) noexcept;
    constexpr cstring_view operator"sv(const char* str, size_t len) noexcept;
    constexpr u8cstring_view operator"sv(const char8_t* str, size_t len) noexcept;
    constexpr u16cstring_view operator"sv(const char16_t* str, size_t len) noexcept;
    constexpr u32cstring_view operator"sv(const char32_t* str, size_t len) noexcept;
    constexpr wcstring_view operator"sv(const wchar_t* str, size_t len) noexcept;
} }
constexpr basic_cstring_view& operator=(const basic_cstring_view&) noexcept = default;
constexpr basic_cstring_view(const charT* str);
constexpr basic_cstring_view(null_terminated_t, const charT* str, size_type len);
constexpr basic_cstring_view(null_terminated_t, const string_view_type&) noexcept;

// [string_view.cstring], iterator support
constexpr const_iterator begin() const noexcept;
constexpr const_iterator end() const noexcept;
constexpr const_iterator cbegin() const noexcept;
constexpr const_iterator cend() const noexcept;
constexpr const_reverse_iterator rbegin() const noexcept;
constexpr const_reverse_iterator rend() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;

friend constexpr const_iterator begin(basic_cstring_view sv) noexcept { return sv.begin(); }
friend constexpr const_iterator end(basic_cstring_view sv) noexcept { return sv.end(); }

// [string_view.cstring], capacity
constexpr size_type size() const noexcept;
constexpr size_type length() const noexcept;
constexpr size_type max_size() const noexcept;
[[nodiscard]] constexpr bool empty() const noexcept;

// [string_view.cstring], element access
constexpr const_reference operator[](size_type pos) const;
constexpr const_reference at(size_type pos) const;
constexpr const_reference front() const;
constexpr const_reference back() const;
constexpr const_pointer data() const noexcept;

constexpr const charT* c_str() const noexcept;

operator basic_string_view< charT, traits>() const noexcept;

// [string_view.cstring], modifiers
constexpr void remove_prefix(size_type n);
constexpr void remove_suffix(size_type n);
constexpr void swap(basic_cstring_view& s) noexcept;

// [string_view.cstring], string operations
constexpr size_type copy(charT* s, size_type n, size_type pos = 0) const;

constexpr basic_cstring_view substr(size_type pos = 0) const;
constexpr string_view_type substr(size_type pos = 0, size_type n) const;

constexpr int compare(string_view_type s) const noexcept;
constexpr int compare(size_type pos1, size_type n1, string_view_type s) const;
constexpr int compare(size_type pos1, size_type n1, string_view_type s, size_type pos2, size_type n2) const;
constexpr int compare(const charT* s) const;
constexpr int compare(size_type pos1, size_type n1, const charT* s) const;
constexpr int compare(size_type pos1, size_type n1, const charT* s, size_type n2) const;

constexpr bool starts_with(string_view_type x) const noexcept;
constexpr bool starts_with(charT x) const noexcept;
constexpr bool starts_with(const charT* x) const;
constexpr bool ends_with_with(string_view_type x) const noexcept;
constexpr bool ends_with_with(charT x) const noexcept;
constexpr bool ends_with_with(const charT* x) const;

// [string_view.cstring], searching
constexpr size_type find(string_view_type s, size_type pos = 0) const noexcept;
constexpr size_type find(charT c, size_type pos = 0) const noexcept;
constexpr size_type find(const charT* s, size_type pos, size_type n) const;
constexpr size_type find(const charT* s, size_type pos = 0) const;
constexpr size_type rfind(string_view_type s, size_type pos = npos) const noexcept;
constexpr size_type rfind(charT c, size_type pos = npos) const noexcept;
constexpr size_type rfind(const charT* s, size_type pos, size_type n) const;
constexpr size_type rfind(const charT* s, size_type pos = npos) const;

constexpr size_type find_first_of(string_view_type s, size_type pos = 0) const noexcept;
constexpr size_type find_first_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type find_first_of(const charT* s, size_type pos = 0) const;
constexpr size_type find_last_of(string_view_type s, size_type pos = npos) const noexcept;
constexpr size_type find_last_of(const charT* s, size_type pos = npos) const noexcept;
constexpr size_type find_last_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type find_last_of(const charT* s, size_type pos = npos) const;

private:
    string_view_type sv_; // exposition only
};
General requirements [string.viewcstring]

Except where otherwise specified in this clause, the requirements on the public member functions of the basic_cstring_view class template are the same as those for the public member function of the same name and function type of basic_string_view. The arguments are forwarded to a member function call of the same name and function type on sv_. If value-returning, the result of the member function call on sv_ is returned.

basic_cstring_view has a class invariant that:

\[
\text{traits::length(sv_.data())} = \text{sv_.size()}
\]

All constructors of basic_cstring_view require arguments such that the constructors effect does not violate that class invariant.

constexpr basic_cstring_view(null_terminated_t, const charT* str, size_type len)

Effects: sv_ is initialized as per sv_(str.len)

constexpr basic_cstring_view(null_terminated_t, const string_view_type& sv) noexcept

Effects: sv_ is initialized by sv

constexpr const charT* c_str() const noexcept;

Returns: data()

constexpr basic_cstring_view substr(size_type pos = 0) const;

Returns: basic_cstring_view(null_terminating, sv_.substr(pos))

constexpr string_view_type substr(size_type pos, size_type n) const;

Returns: sv_.substr(pos, n)

operator basic_string_view<charT, traits>() const noexcept;

Returns: sv_