Document Number: P0254R1
Date: 2016-05-29
Audience: Library Evolution Working Group
Reply to: Marshall Clow <marshall@idio.com>

Integrating `std::string_view` and `std::string`

Basic Approach

In the Library Fundamentals Technical Specification (LFTS), we introduced `std::experimental::string_view` (henceforth `string_view`), and it has proven to be very popular. In Jacksonville, it was approved for C++17. I believe that there are some changes that should be made to better integrate it into the standard library.

When `string_view` was proposed, one of the constraints put upon it (being part of the LFTS) was that no changes could be made to existing classes in the standard library. Where changes were deemed necessary, (function, for example) the components were duplicated in the LFTS, and the changes made there.

The upshot of this was that the connection between `std::string` (henceforth `string`) and `string_view` was all done in `string_view`. `string_view` has:

* An implicit conversion from `string`
* a member function `to_string`, which creates a new `string`.

I believe that this is backwards; that `string_view` should know nothing of `string`, and that `string` should handle the conversions between the types. Specifically, `string` should have:

* An implicit conversion to `string_view`
* An explicit constructor from a `string_view`.

Rationale

* `string_view` as a basic vocabulary type leads to additional efficiencies.

Because it does not own the underlying data, a `string_view` is cheap to construct and to copy. The guidance that we give is that these should be passed by value. When there are no lifetime issues (and where null-termination is not an issue), `string_view` is a superior vocabulary type than `string`, and the standard should prefer it.

For example, there are several member functions of `string` (`find, rfind,`

find_first_of, find_last_of, etc) that are defined in terms of creating a temporary string. It would be much more efficient to create a string_view instead.

Given:

```
     void foo ( const string & blah ) { /* do something with
blah */ }
```

calling it as:

```
     foo ( “Supercalifragilisticexpialidocious” );
```

requires a call to traits::length, a memory allocation, a call to memcpy, and then (after the call returns) a memory deallocation. Memory allocation is not cheap, and in a multithreaded environment must be protected against data races.

However, if we write instead:

```
     void foo ( string_view blah ) { /* do something with blah
*/ }
```

then the same call requires only a call to traits::length.

Creating a string_view from a string is cheap, hence the implicit conversion. Creating a string from a string_view is not cheap, so it should be explicit.

* Support for other string types.

Currently, we have a single string type in the standard library: std::string. Users have many of their own QString, CString, along with innumerable home-grown versions. Using them with the rest of the standard library is currently a pain point for users. If they store their data in contiguous memory, they can support string_view. If the standard library uses string_view widely, they could use their string type with standard library routines.

Consider outputting data from a homegrown string class (for purposes of exposition, called home_string). Implementing operator<< is a fair amount of work, requiring a reasonably complete knowledge of the entire iostreams infrastructure. On the other hand, with string_view, someone could write:

```
template<class charT, class traits, class Allocator>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os,
               const home_string<charT,traits,Allocator>& str)
    {
    return os << string_view<charT, traits>(str);
    }
```

and get all the formatting, etc “for free”. They still have to write an extraction operator, but that is much simpler than insertion.

Wording

All changes are relative to N4582]

In [string.view.template], remove:

```
     basic_string_view(const basic_string<charT, traits,
Allocator>& str) noexcept;

     // 7.8, basic_string_view string operations
     template<class Allocator>
     explicit operator basic_string<charT, traits, Allocator>()
const;
     template<class Allocator = allocator<charT> >
basic_string<charT, traits, Allocator> to_string(
       const Allocator& a = Allocator()) const;
```

In [string.view.cons], remove:

```
    template<class Allocator>
    basic_string_view(const basic_string<charT, traits,
Allocator>& str) noexcept;
```

3. Effects: Constructs a `basic_string_view`, with the postconditions in table 75.

and remove Table 75.

In [string.view.ops], remove:

```
    template<class Allocator>
    explicit operator basic_string<charT, traits, Allocator>()
const;
```

1. *Effects*: Equivalent to `return basic_string<charT, traits, Allocator>(begin(), end());`
2. *Complexity*: O(`size()`)
3. [*Note*: Users who want to control the allocator instance should call `to_string(allocator)`. — end note]

```
    template<class Allocator = allocator<charT>>
      basic_string<charT, traits, Allocator>
        to_string(const Allocator& a = Allocator()) const;
```

4. *Returns*: `basic_string<charT, traits, Allocator>(begin(), end(), a)`.
5. *Complexity*: O(`size()`)

In [basic.string], add:

```
    basic_string(const basic_string& str, size_type pos,
size_type n,
                 const Allocator& a = Allocator());
    explicit basic_string(basic_string_view<charT, traits> sv,
                          const Allocator& a = Allocator());
    basic_string(const charT* s,
                 size_type n, const Allocator& a = Allocator());

...
    basic_string& operator=(initializer_list<charT>);
    basic_string& operator=(basic_string_view<charT, traits> sv);
    operator basic_string_view<charT, traits>() const;

...
    basic_string& append(const basic_string& str, size_type pos,
                         size_type n = npos);
    basic_string& append(basic_string_view<charT, traits> sv);
    basic_string& append(basic_string_view<charT, traits> sv,
size_type pos,
                         size_type n = npos);

...
    basic_string& operator+=(const basic_string& str);
    basic_string& operator+=(basic_string_view<charT, traits>
sv);

...
    basic_string& assign(const basic_string& str,
                         size_type pos, size_type n = npos);
    basic_string& assign(basic_string_view<charT, traits> sv);
    basic_string& assign(basic_string_view<charT, traits> sv,
                         size_type pos, size_type n = npos);

...
    basic_string& insert(size_type pos1, const basic_string& str,
                         size_type pos2, size_type n = npos);
    basic_string& insert(size_type pos1, basic_string_view<charT,
traits> sv);
    basic_string& insert(size_type pos1, basic_string_view<charT,
traits> sv,
                         size_type pos2, size_type n = npos);

...
```

```
    basic_string& replace(size_type pos1, size_type n1,
                          const basic_string& str,
                          size_type pos2, size_type n2 = npos);

    basic_string& replace(size_type pos1, size_type n1,
                          basic_string_view<charT, traits> sv);
    basic_string& replace(size_type pos1, size_type n1,
                          basic_string_view<charT, traits> sv,
                          size_type pos2, size_type pos = npos);

...
    basic_string& replace(const_iterator i1, const_iterator i2,
                          const basic_string& str);
    basic_string& replace(const_iterator i1, const_iterator i2,
                          basic_string_view<charT, traits> sv);

...
    size_type find (const basic_string& str,
                    size_type pos = 0) const noexcept;
    size_type find (basic_string_view<charT, traits> sv,
                    size_type pos = 0) const noexcept;

...
    size_type rfind(const basic_string& str,
                    size_type pos = npos) const noexcept;
    size_type rfind(basic_string_view<charT, traits> sv,
                    size_type pos = npos) const noexcept;

...
    size_type find_first_of(const basic_string& str,
                            size_type pos = 0) const noexcept;
    size_type find_first_of(basic_string_view<charT, traits> sv,
                            size_type pos = 0) const noexcept;

...
    size_type find_last_of (const basic_string& str,
                            size_type pos = npos) const
noexcept;
    size_type find_last_of (basic_string_view<charT, traits> sv,
                            size_type pos = npos) const
noexcept;

...
    size_type find_first_not_of(const basic_string& str,
                size_type pos = 0) const noexcept;
    size_type find_first_not_of(basic_string_view<charT, traits>
sv,
```

```
                size_type pos = 0) const noexcept;

...
    size_type find_last_not_of (const basic_string& str,
                                size_type pos = npos) const
noexcept;
    size_type find_last_not_of (basic_string_view<charT, traits>
sv,
                                size_type pos = npos) const
noexcept;

    int compare(size_type pos1, size_type n1,
                const basic_string& str,
                size_type pos2, size_type n2 = npos) const;

    int compare(basic_string_view<charT, traits> sv) const
noexcept;
    int compare(size_type pos1, size_type n1,
                basic_string_view<charT, traits> sv) const;
    int compare(size_type pos1, size_type n1,
                basic_string_view<charT, traits> sv,
                size_type pos2, size_type n2 = npos) const;
```

In [string.cons], add:

```
    explicit basic_string(basic_string_view<charT, traits> sv,
const Allocator& a = Allocator());
```
Effects: Same as `basic_string(sv.begin(), sv.size(), a)`.

```
    basic_string& operator=(basic_string_view<charT, traits> sv);
```
Returns: `*this = basic_string(sv)`.

Add a new section [string.operator], after [string.cons]:

```
    operator basic_string_view<charT, traits>() const;
```
Effects: equivalent to `return basic_string_view<charT, traits>(data(), size())`.

In [string.append], add:

basic_string&
operator+=(basic_string_view<charT, traits> sv);
Effects: Calls `append(sv)`.
Returns: `*this`.

```
basic_string& append(basic_string_view<charT, traits> sv);
```
Effects: equivalent to `return append(sv.data(), sv.size())`.

```
basic_string& append(basic_string_view<charT, traits> sv,
                     size_type pos, size_type n = npos);
```
Throws: `out_of_range` if `pos > sv.size()`.
Effects: Determines the effective length `rlen` of the string to append as the smaller of `n` and `sv.size() - pos` and calls `append(str.data() + pos, rlen)`.
Returns: `*this`.

In [string.assign], add:

```
basic_string& assign(basic_string_view<charT, traits> sv);
```
Effects: Equivalent to `return assign(sv.data(), sv.size())`.

```
basic_string&
  assign(basic_string_view<charT, traits> sv,
         size_type pos, size_type n = npos);
```
Throws: `out_of_range` if `pos > sv.size()`.
Effects: Determines the effective length `rlen` of the string to assign as the smaller of `n` and `sv.size() - pos` and calls `assign(sv.data() + pos, rlen)`.
Returns: `*this`.

In [string.insert], add:

```
basic_string& insert(size_type pos1,
                     basic_string_view<charT, traits> sv);
```
Effects: Equivalent to `return insert(pos1, sv.data(), sv.size())`.

```
basic_string& insert(size_type pos1,
                     basic_string_view<charT, traits> sv,
                     size_type pos2, size_type pos = npos);
```
Throws: `out_of_range` if `pos1 > size()` or `pos2 > sv.size()`.
Effects: Determines the effective length `rlen` of the string to assign as the smaller of `n` and `sv.size() - pos2` and calls `insert(pos1, sv.data() + pos2, rlen)`.
Returns: `*this`.

In [string.replace], add:

```
basic_string& replace(size_type pos1, size_type n1,
                     basic_string_view<charT, traits> sv);
```

Effects: Equivalent to `return replace(pos1, n1, sv.data(), sv.size());`

```
basic_string& replace(size_type pos1, size_type n1,
                      basic_string_view<charT, traits> sv,
                      size_type pos2, size_type pos = npos);
```

Throws: `out_of_range` if `pos1 > size()` or `pos2 > sv.size()`.
Effects: Determines the effective length `rlen` of the string to be inserted as the smaller of `n2` and `sv.size() - pos2` and calls `replace(pos1, n1, str.data() + pos2, rlen)`.
Returns: `*this`.

```
basic_string& replace(const_iterator i1, const_iterator i2,
                      basic_string_view<charT, traits> sv);
```

Requires: `[begin(), i1)` and `[i1, i2)` are valid ranges.
Effects: Calls `replace(i1 - begin(), i2 - i1, sv)`.
Returns: `*this`.

In [string.find], add:

```
size_type find (basic_string_view<charT, traits> sv,
                size_type pos = 0) const noexcept;
```

Effects: Determines the lowest position `xpos`, if possible, such that both of the following conditions obtain:
— `pos <= xpos` and `xpos + sv.size() <= size()`;
— `traits::eq(at(xpos+I), sv.at(I))` for all elements `I` of the data referenced by `sv`.
Returns: `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.
Remarks: Uses `traits::eq()`.

and change:

```
size_type find (const basic_string& str,
                size_type pos = 0) const noexcept;
```

Effects: equivalent to return `find(basic_string_view<charT, traits>(str), pos)`.
Effects: Determines the lowest position `xpos`, if possible, such that both of the following conditions obtain:
— `pos <= xpos` and `xpos + sv.size() <= size()`;
— `traits::eq(at(xpos+I), str.at(I))` for all elements `I` of the string controlled by str.
Returns: `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.
Remarks: Uses `traits::eq()`.

`size_type find(const charT* s, size_type pos, size_type n) const;`

Returns: `find(basic_string_view<CharT, traits>(s,n), pos)`.

`size_type find(const charT* s, size_type pos = 0) const;`

Requires: `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

Returns: `find(basic_string_view<CharT, traits>(s), pos)`.

In [string.rfind], add:

`size_type rfind (basic_string_view<charT, traits> sv, size_type pos = 0) const noexcept;`

Effects: Determines the highest position `xpos`, if possible, such that both of the following conditions obtain:

— `pos <= xpos` and `xpos + sv.size() <= size()`;

— `traits::eq(at(xpos+I), sv.at(I))` for all elements `I` of the data referenced by `sv`.

Returns: `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

Remarks: Uses `traits::eq()`.

and change:

`size_type rfind (const basic_string& str, size_type pos = 0) const noexcept;`

Effects: equivalent to return `rfind(basic_string_view<charT, traits>(str), pos)`.

Effects: Determines the highest position `xpos`, if possible, such that both of the following conditions obtain:

— `pos <= xpos` and `xpos + sv.size() <= size()`;

— `traits::eq(at(xpos+I), str.at(I))` for all elements `I` of the string controlled by str.

Returns: `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

Remarks: Uses `traits::eq()`.

`size_type rfind(const charT* s, size_type pos, size_type n) const;`

Returns: `rfind(basic_string_view<CharT, traits>(s,n), pos)`.

`size_type rfind(const charT* s, size_type pos = 0) const;`

Requires: `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

Returns: `rfind(basic_string_view<CharT, traits>(s),`

pos).

In [string.find.first.of], add:

```
        size_type find_first_of (basic_string_view<charT, traits>
sv, size_type pos = 0) const noexcept;
```

Effects: Determines the lowest position xpos, if possible, such that both of the following conditions obtain:

— pos <= xpos and xpos < size();

— traits::eq(at(xpos), sv.at(I)) for some element I of the data referenced by sv.

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

Remarks: Uses traits::eq().

and change:

```
        size_type find_first_of (const basic_string& str,
size_type pos = 0) const noexcept;
```

Effects: equivalent to return find_first_of(basic_string_view<charT, traits>(str), pos).

Effects: Determines the lowest position xpos, if possible, such that both of the following conditions obtain:

— pos <= xpos and xpos < size();

— traits::eq(at(xpos), str.at(I)) for some element I of the string controlled by str.

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

Remarks: Uses traits::eq().

```
        size_type find_first_of(const charT* s, size_type pos,
size_type n) const;
```

Returns: find_first_of(basic_string_view<CharT, traits>(s,n), pos).

```
        size_type find_first_of(const charT* s, size_type pos =
0) const;
```

Requires: s points to an array of at least traits::length(s) + 1 elements of charT.

Returns: find_first_of(basic_string_view<CharT, traits>(s), pos).

In [string.find.last.of], add:

```
        size_type find_last_of (basic_string_view<charT, traits>
sv, size_type pos = 0) const noexcept;
```

Effects: Determines the highest position xpos, if possible, such that

both of the following conditions obtain:

— `pos <= xpos and xpos < size();`

— `traits::eq(at(xpos), sv.at(I))` for some element `I` of the data referenced by `sv`.

Returns: `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

Remarks: Uses `traits::eq()`.

and change:

```
size_type find_last_of (const basic_string& str,
size_type pos = 0) const noexcept;
```

Effects: equivalent to return `find_first_of(basic_string_view<charT, traits>(str), pos)`.

Effects: Determines the highest position `xpos`, if possible, such that both of the following conditions obtain:

— `pos <= xpos and xpos < size();`

— `traits::eq(at(xpos), str.at(I))` for some element `I` of the string controlled by str.

Returns: `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

Remarks: Uses `traits::eq()`.

```
size_type find_last_of(const charT* s, size_type pos,
size_type n) const;
```

Returns: `find_last_of(basic_string_view<CharT, traits>(s,n), pos)`.

```
size_type find_last_of(const charT* s, size_type pos = 0)
const;
```

Requires: `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

Returns: `find_last_of(basic_string_view<CharT, traits>(s), pos)`.

In [string.find.first.not.of], add:

```
size_type find_first_not_of (basic_string_view<charT,
traits> sv, size_type pos = 0) const noexcept;
```

Effects: Determines the lowest position `xpos`, if possible, such that both of the following conditions obtain:

— `pos <= xpos and xpos < size();`

— `traits::eq(at(xpos), sv.at(I))` for no element `I` of the data referenced by `sv`.

Returns: `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

Remarks: Uses `traits::eq()`.

and change:

`size_type find_first_not_of (const basic_string& str, size_type pos = 0) const noexcept;`

Effects: equivalent to return `find_first_not_of(basic_string_view<charT, traits>(str), pos)`.

Effects: Determines the lowest position `xpos`, if possible, such that both of the following conditions obtain:

— `pos <= xpos` and `xpos + sv.size() <= size()`;

— `traits::eq(at(xpos), str.at(I))` for no element `I` of the string controlled by str.

Returns: `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

Remarks: Uses `traits::eq()`.

`size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;`

Returns: `find_first_not_of(basic_string_view<CharT, traits>(s,n), pos)`.

`size_type find_first_not_of(const charT* s, size_type pos = 0) const;`

Requires: `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

Returns: `find_first_not_of(basic_string_view<CharT, traits>(s), pos)`.

In [string.find.last.not.of], add:

`size_type find_last_not_of (basic_string_view<charT, traits> sv, size_type pos = 0) const noexcept;`

Effects: Determines the highest position `xpos`, if possible, such that both of the following conditions obtain:

— `pos <= xpos and xpos < size()`;

— `traits::eq(at(xpos), sv.at(I))` for no element `I` of the data referenced by `sv`.

Returns: `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

Remarks: Uses `traits::eq()`.

and change:

`size_type find_last_not_of (const basic_string& str, size_type pos = 0) const noexcept;`

Effects: equivalent to return `find_last_not_of(basic_string_view<charT, traits>(str), pos)`.

Effects: Determines the highest position `xpos`, if possible, such that both of the following conditions obtain:

— `pos <= xpos` and `xpos + sv.size() <= size()`;
— `traits::eq(at(xpos), str.at(I))` for no element `I` of the string controlled by str.

Returns: `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

Remarks: Uses `traits::eq()`.

```
size_type find_last_not_of(const charT* s, size_type pos,
size_type n) const;
```

Returns: `find_last_not_of(basic_string_view<CharT, traits>(s,n), pos)`.

```
size_type find_last_not_of(const charT* s, size_type pos
= 0) const;
```

Requires: `s` points to an array of at least `traits::length(s) + 1` elements of `charT`.

Returns: `find_last_not_of(basic_string_view<CharT, traits>(s), pos)`.

In [string.compare], add:

```
int compare(basic_string_view<charT, traits> sv) const
noexcept;
```

Effects: Determines the effective length `rlen` of the strings to compare as the smallest of `size()` and `sv.size()`. The function then compares the two strings by calling `traits::compare(data(), sv.data(), rlen)`.

Returns: The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table XX.

[Editor's note: Add new table similar to table 74, but with 'sv' where table 74 shows 'str']

```
int compare(size_type pos1, size_type n1,
            basic_string_view<charT, traits> sv) const;
```

Effects: Equivalent to `return basic_string_view<charT, traits>(this.data(), pos1, n1).compare(sv)`.

```
int compare(size_type pos1, size_type n1,
            basic_string_view<charT, traits> sv,
            size_type pos2, size_type n2 = npos) const;
```

Effects: Equivalent to `return basic_string_view<charT, traits>(this.data(), pos1, n1).compare(sv, pos2, n2)`.

and change:

```
int compare(const basic_string& str) const noexcept;
```

Effects: Determines the effective length `rlen` of the strings to compare as the smallest of `size()` and `sv.size()`. The function then compares the two strings by

calling `traits::compare(data(), sv.data(), rlen)`.

Returns: The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 74.

Effects: Equivalent to `return compare(basic_string_view<charT, traits>(str))`.

[Editor's note: Remove table 74]

```
    int compare(size_type pos1, size_type n1,
                const basic_string& str) const;
```

Returns: `basic_string(*this,pos1,n1).compare(str)`.

Effects: equivalent to `return compare(pos1, n1, basic_string_view<CharT, traits>(str))`.

```
    int compare(size_type pos1, size_type n1,
                const basic_string& str,
                size_type pos2, size_type n2 = npos) const;
```

Returns: `basic_string(*this, pos1, n1).compare(basic_string(str, pos2, n2))`.

Effects: equivalent to `return compare(pos1, n1, basic_string_view<CharT, traits>(str), pos2, n2)`.

Implementation Status

I have implemented most of this in libc++ (on a branch). I have not implemented `basic_string::find, find_first_of` or `find_last_of`, `find_first_not_of, find_last_not_of` and `compare`, but have implemented all the other proposed changes.

The resulting library passes all of its tests, and successfully builds boost as well.

Future work

[Note: I am NOT proposing any of these changes at the current time. They can be added later, but the changes in the relationship between `string` and `string_view` will be difficult to change once we ship them in their current state.]

There are a lot of calls in the standard library that take strings as parameters. Some of these can be changed to take a string_view, and due to the implicit conversion, user code should continue to work (after a recompilation).

Example:

std::logic_error and std::runtime_error (and each of their subclasses) have two constructors:

```
explicit logic_error(const string& what_arg);
explicit logic_error(const char* what_arg);
```

which immediately copy the data into a member variable. These could be replaced with a single constructor:

```
explicit logic_error(string_view what_arg);
```

The codecvt facilities [conversions.string] all take input parameters as both const char * and string (or wchar_t and string). Those could be string_views.
Other possibilities include:

* bitset has a constructor from a string
* Locale's constructor takes a string/const char *, and "name" returns a string.
* ctype_byname has two constructors that take a string/const char *
* the various locale::facet subclasses could return string_refs instead of string.
* There's a lot of opportunities in <regex>.

On the other hand, there are many calls in the standard library that take strings as parameters, and then pass them on to the underlying OS, which expects a null-terminated string. In general, I am NOT proposing that we replace those calls with a `string_view` version, because that would require allocating memory and copying the data, and the whole point of string_view is to not do that when we don't have to.

Example:

std::basic_ifstream and basic_ofstream (and each of their subclasses) have two constructors:

```
explicit basic_ifstream(const char* s,
  ios_base::openmode mode = ios_base::in);
explicit basic_ifstream(const string& s,
  ios_base::openmode mode = ios_base::in);
```

There are a several functions/classes in the standard library that mutate strings. They are NOT candidates for using string_view
Examples:

`basic_stringbuf/basic_istringstream/basic_ostringstream`