Document Number: P0254R0
Date: 2016-03-20
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.

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 than insertion.

List of proposed changes

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;
```

Remove [string.view.cons] paragraphs 4, 5, and table 8
Remove [string.view.ops] paragraphs 1 through 7.

In 21.4 [basic.string]:

```
    namespace std {
      template<class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
      class basic_string {
          public:
```

. . .

```
     explicit basic_string(basic_string_view<charT, traits> sv, const
Allocator& a = Allocator());
     operator basic_string_view<charT, traits>() const;
```

The rest of these are not strictly necessary, but they would be useful (i.e, more efficient).

```
     basic_string& assign(basic_string_view<charT, traits> sv);
     basic_string& assign(basic_string_view<charT, traits> sv,
size_type n, size_type pos = 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 pos = 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,
basic_string_view<charT, traits> sv);
```

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

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

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

. . .

Wording

I don't have any proposed wording at this time, because we don't have a draft standard with both `string` and `string_view` in it. I will provide wording before Oulu unless discussion indicates that there is no consensus for making this change.

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`, but have implemented all the other proposed changes.

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

Future work

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
```