fixed_string: a compile-time string

Document No.: P0259R0
Revises: N4121[1], N4236[2]
Project: Programming Language C++
Audience: Library Evolution
Authors: Michael Price <michael.b.price.dev@gmail.com>
Andrew Tomazos <andrewtomazos@gmail.com>

Summary

We propose a class template `basic_fixed_string` for inclusion in the Library Fundamentals Technical Specification.

The main purpose for the template is to provide a string type that is usable in constexpr programming. `fixed_string` is constexpr-compatible, `std::string` is not.

For example:

```
constexpr auto username = std::make_fixed_string("fred");
constexpr auto home_dir = "/home/" + username;
static_assert(home_dir == "/home/fred");
```

The above concatenation and comparison are guaranteed to occur at compile-time. They entail no heap allocations and zero run-time code.

As a constexpr-compatible type (a “literal type” in standardese), a `fixed_string` may be:

- the type of a constexpr object
- the parameter type of a constexpr function.
- the return type of a constexpr function
- the type of a local variable of a constexpr function. (Such local variables may even be modified.)

`std::string` has none of these properties. The following does not work:

---

[1] "Compile-Time String: std::string_literal<n>." 2014. 11 Feb. 2016 <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4121.pdf>

[2] "N4236 - A compile-time string library template with UDL operator templates ." 2014. 11 Feb. 2016 <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4236.html>

```
constexpr std::string username = "fred"; // ERROR
```

The reason is that std::`string` is not a constant expression-friendly type. It possibly allocates memory dynamically. The possibility of extending the core language rules to enable this has been explored and rejected by EWG.

Why not just use `std::experimental::string_view`[3]?

`string_view` is **non-owning**, which means that it is unsuitable to act as the return type of a newly formed string. For example, there is no way to reasonably implement the following function:

```
// concatenate two strings
constexpr string_view concat(string_view a, string_view b);
```

There is nowhere to give ownership of the newly formed string.

Why not just use `std::array<char>`?

`array<char>` does not have a string-like interface. `fixed_string` has an interface that is consistent with `string` and `string_view`, and that is designed for text handling.

`array<char>` doesn't maintain a null-terminated invariant. To append a null-terminator one would need to allocate storage and copy the text to there. `fixed_string` on the other hand offers a `constexpr noexcept c_str()` constant-time member function.

Why not use a `char` parameter pack ?

This technique was proposed in N4236 and was received unfavorably in EWG. `char` parameter packs are practically limited in length (1024 in some implementations) and perform poorly compile-time and link-time compared to `fixed_string`. This is because `fixed_string` stores the text as subobjects (like an array) and not as template parameters. `fixed_string` only has the length as a template parameter. If a parameter pack is truly needed it is possible to convert a `constexpr fixed_string` to a `char` parameter pack, and vice versa, as shown in N4121.

Why not use built-in string literals?

Built-in string literals are not types. The type of a built-in string literal is a built-in array, which have an even worse interface than std::array<char> (see above) for this purpose. For example:

---

[3] "N4564 - Programming Languages — C++ Extensions for Library Fundamentals, Version 2 PDTS." 2015. 11 Feb. 2016 <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4564.pdf>

```
constexpr auto username = "fred"; // OK
constexpr auto home_dir = "/home/" + username; // WRONG
static_assert(home_dir == "/home/fred"); // WRONG
```

Built-in string literals cannot be computed at compile-time. They are tokens, set in stone after preprocessing.

Motivations

The type model of literal strings in C++ was largely inherited directly from C, with a few minor tweaks throughout the years (such as restricting conversions to non-const `char*`). The inclusion of the standard library's `std::basic_string` and std::basic_string_view templates solve many of the problems with the usability of string literals by providing a rich interface for accessing, searching, and manipulating the value of the string. There are many places where `std::basic_string`, `std::basic_string_view`, and `const char*` all fall short.

```
// Concatenation
///
auto x1 = "Hello" + ", " + "World!"; // Error!
auto x2 = std::string("Hello") + ", " + "World!"; // Valid, but
                                                  // odd. Also
                                                  // runtime cost.
auto x3 = "Hello" + std::string(", ") + "World!"; // Error!
auto x4 = "Hello" ", " "World!"; // Valid, but...

auto conjunction = std::string(", ");
auto x5 = "Hello" conjuction "World!"; // Error!



// Generic Programming
//
template <typename Str>
auto format(Str s)
{
    return s;
}

template <typename Str, typename T, typename ... Args>
auto format(Str s, T&& t, Args&&... args)
{
    auto pos = s.find('%');
```

```
    auto front = s.substr(0, pos);
    auto back = s.substr(pos+1);
    return front + t + format(back, args...);
}

auto s1 = format("%, %!", "Hello", "World"); // Error!
auto s2 = format("%, %!"s, "Hello", "World"); // Okay, but runtime
auto s2 = format(string_view("%, %!"), "Hello", "World"); // Error!
```

With a compile-time string type, we can make these calculations occur at compile-time easily.

```
#define S(s) make_fixed_string(s);

// Concatenation
///
constexpr auto x1 = S("Hello") + ", " + "World!"; // Works great!

constexpr auto conjunction = S(", ");
constexpr auto x5 = S("Hello") + conjunction + "World!"; // Great!



// Generic Programming
//
template <typename Str>
constexpr auto format(Str s)
{
    return s;
}

template <typename Str, typename T, typename ... Args>
constexpr auto format(Str s, T&& t, Args&&... args)
{
    auto pos = s.find('%');
    auto front = s.substr(0, pos);
    auto back = s.substr(pos+1);
    return front + t + format(back, args...);
}

constexpr auto s1 = format(S("%, %!"), "Hello", "World"); // Voila!
```

Usages

A compile-time string utility can be useful in multiple fields:

- Metaprogramming - See the many existing implementations[4] [5] [6] [7] [8] [9] [10].
- Reflection - Easily synthesizing new members at compile-time.
- Localization - Usually involves string formatting, and sometimes done at compile-time.
- Library design - Could be used to form more useful `static_assert` messages[11].

[4] "Conveniently Declaring Compile-Time Strings in C++ ..." 2013. 11 Feb. 2016 <http://stackoverflow.com/questions/15858141/conveniently-declaring-compile-time-strings-in-c>
[5] "Chapter 21. Boost.Metaparse - develop." 2015. 11 Feb. 2016 <http://www.boost.org/doc/libs/develop/doc/html/metaparse.html>
[6] "Parsing strings at compile-time — Part I | Andrzej's C++ blog." 2011. 11 Feb. 2016 <https://akrzemi1.wordpress.com/2011/05/11/parsing-strings-at-compile-time-part-i/>
[7] "constexprstr." 2011. 11 Feb. 2016 <http://constexprstr.svn.sourceforge.net/viewvc/constexprstr/main.cpp?revision=9&view=markup>
[8] "Template Programming Compile Time String Functions." 2015. 11 Feb. 2016 <http://accu.org/index.php/journals/2137>
[9] "GitHub - irrequietus/typestring: C++11/14 strings for direct ..." 2015. 11 Feb. 2016 <https://github.com/irrequietus/typestring>
[10] "C++11 Compile-time String Concatenation with constexpr." 2015. 11 Feb. 2016 <https://www.daniweb.com/programming/software-development/code/482276/c-11-compile-time-string-concatenation-with-constexpr>
[11] "N4433 - Flexible static_assert messages." 2015. 11 Feb. 2016 <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4433.html>

Partial Specification

Constant expression string classes [fixed_string.classes]

The header `<experimental/fixed_string>` defines the `basic_fixed_string` class template for interacting, possibly at compile-time, with fixed-length sequences of char-like objects and four type alias templates, `fixed_string<N>`, `fixed_u16string<N>`, `fixed_u32string<N>`, and `fixed_wstring<N>`, that name the specializations `basic_fixed_string<char,N>`, `basic_fixed_string<char16_t,N>`, `basic_fixed_string<char32_t,N>`, and `basic_fixed_string<wchar_t,N>`, respectively.

Header `<experimental/fixed_string>` synopsis

```
namespace std {
namespace experimental {
inline namespace fundamentals_vXXXX {

// Class template basic_fixed_string
template <class charT, size_t N>
  class basic_fixed_string;

// basic_fixed_string non-member concatenation functions
template <class charT, size_t L, size_t R>
  constexpr basic_fixed_string<charT, N + M>
    operator+(const basic_fixed_string<charT, L>& lhs,
              const basic_fixed_string<charT, R>& rhs) noexcept;

template <class charT, size_t L, size_t R>
  constexpr basic_fixed_string<charT, N - 1 + M>
    operator+(const charT(&lhs)[L],
              const basic_fixed_string<charT, R>& rhs) noexcept;

template <class charT, size_t N>
  constexpr basic_fixed_string<charT, N + 1>
    operator+(charT lhs,
              const basic_fixed_string<charT, R>& rhs) noexcept;

template <class charT, size_t L, size_t R>
  constexpr basic_fixed_string<charT, N + M - 1>
    operator+(const basic_fixed_string<charT, L>& lhs,
              const charT(&rhs)[R]) noexcept;
```

```
template <class charT, size_t N>
  constexpr basic_fixed_string<charT, N + 1>
    operator+(const basic_fixed_string<charT, L>& lhs,
              charT rhs) noexcept;


// basic_fixed_string non-member comparison functions
template <class charT, size_t L, size_t R>
  constexpr bool operator==(const basic_fixed_string<charT, L>& lhs,
                            const basic_fixed_string<charT, R>& rhs)
    noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator==(const charT(&lhs)[L],
                            const basic_fixed_string<charT, R>& rhs)
    noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator==(const basic_fixed_string<charT, L>& lhs,
                            const charT(&rhs)[R]) noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator!=(const basic_fixed_string<charT, L>& lhs,
                            const basic_fixed_string<charT, R>& rhs)
    noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator!=(const charT(&lhs)[L],
                            const basic_fixed_string<charT, R>& rhs)
    noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator!=(const basic_fixed_string<charT, L>& lhs,
                            const charT(&rhs)[R]) noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator< (const basic_fixed_string<charT, L>& lhs,
                           const basic_fixed_string<charT, R>& rhs)
    noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator< (const charT(&lhs)[L],
                            const basic_fixed_string<charT, R>& rhs)
    noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator< (const basic_fixed_string<charT, L>& lhs,
                            const charT(&rhs)[R]) noexcept;
```

```
template <class charT, size_t L, size_t R>
  constexpr bool operator> (const basic_fixed_string<charT, L>& lhs,
                            const basic_fixed_string<charT, R>& rhs)
    noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator> (const charT(&lhs)[L],
                            const basic_fixed_string<charT, R>& rhs)
    noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator> (const basic_fixed_string<charT, L>& lhs,
                            const charT(&rhs)[R]) noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator<=(const basic_fixed_string<charT, L>& lhs,
                            const basic_fixed_string<charT, R>& rhs)
    noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator<=(const charT(&lhs)[L],
                            const basic_fixed_string<charT, R>& rhs)
    noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator<=(const basic_fixed_string<charT, L>& lhs,
                            const charT(&rhs)[R]) noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator>=(const basic_fixed_string<charT, L>& lhs,
                            const basic_fixed_string<charT, R>& rhs)
    noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator>=(const charT(&lhs)[L],
                            const basic_fixed_string<charT, R>& rhs)
    noexcept;

template <class charT, size_t L, size_t R>
  constexpr bool operator>=(const basic_fixed_string<charT, L>& lhs,
                            const charT(&rhs)[R]) noexcept;

// swap
template <class charT, size_t N>
  constexpr void swap(basic_fixed_string<charT, L>& lhs,
                      basic_fixed_string<charT, N>& rhs) noexcept;
```

```
// basic_fixed_string type aliases
template <size_t N>
  using fixed_string = basic_fixed_string<char, N>;
template <size_t N>
  using fixed_u16string = basic_fixed_string<char16_t, N>;
template <size_t N>
  using fixed_u32string = basic_fixed_string<char32_t, N>;
template <size_t N>
  using fixed_wstring = basic_fixed_string<wchar_t, N>;


// numeric conversions:
template <size_t N>
  constexpr int stoi(const fixed_string<N>& str, int base = 10);

template <size_t N>
  constexpr unsigned stou(const fixed_string<N>& str, int base = 10);

template <size_t N>
  constexpr long stol(const fixed_string<N>& str, int base = 10);

template <size_t N>
  constexpr unsigned long stoul(const fixed_string<N>& str, int base = 10;

template <size_t N>
  constexpr long long stoll(const fixed_string<N>& str, int base = 10);

template <size_t N>
  constexpr unsigned long long stoull(const fixed_string<N>& str,
                                      int base = 10);

template <size_t N>
  constexpr float stof(const fixed_string<N>& str);

template <size_t N>
  constexpr double stod(const fixed_string<N>& str);

template <size_t N>
  constexpr long double stold(const fixed_string<N>& str);

template <int val>
  constexpr fixed_string</*...*/> to_fixed_string_i() noexcept;

template <unsigned val>
  constexpr fixed_string</*...*/> to_fixed_string_u() noexcept;

template <long val>
  constexpr fixed_string</*...*/> to_fixed_string_l() noexcept;
```

```
template <unsigned long val>
  constexpr fixed_string</*...*/> to_fixed_string_ul() noexcept;

template <long long val>
  constexpr fixed_string</*...*/> to_fixed_string_ll() noexcept;

template <unsigned long long val>
  constexpr fixed_string</*...*/> to_fixed_string_ull() noexcept;

// XY.N+6, creation helper function
template <class charT, size_t N>
  constexpr basic_fixed_string<charT, N - 1>
    make_fixed_string(const charT(&a)[N]) noexcept;

}
}
}
```

Class template `basic_fixed_string` [basic.fixed_string]

The class template `basic_fixed_string` describes objects that store a sequence of a fixed number of arbitrary char-like objects with the first element of the sequence at position zero. They are initialized with constant expressions and are stored in the program image. Such a sequence is also called a “compile-time string” if the type of the char-like objects that it holds is clear from context. In the rest of this Clause, the type of the char-like objects held in a `basic_fixed_string` object is designated by `charT`, and the number of stored objects is designated by `N`.

A `fixed_basic_string` is a contiguous container (23.2.1).

```
namespace std {
namespace experimental {
inline namespace fundamentals_vXXXX {

  template <class charT, size_t N>
  class basic_fixed_string {
  public:
    // types:
    using value_type = charT;

    using view = basic_string_view<charT>;
    using size_type = view::size_type;
    using difference_type = view::difference_type;

    using reference       = value_type&;
    using const_reference = const value_type&;
```

```
using pointer         = value_type*;
using const_pointer   = const value_type*;

using iterator               = pointer;
using const_iterator         = const_pointer;
using reverse_iterator       = std::reverse_iterator<iterator>;
using const_reverse_iterator = std::reverse_iterator<const_iterator>;
static constexpr size_type npos = view::npos;

// construct/copy/conversions
constexpr basic_fixed_string() noexcept;
constexpr basic_fixed_string(const basic_fixed_string& str) noexcept;
constexpr basic_fixed_string(const charT(&arr)[N + 1]) noexcept;

constexpr basic_fixed_string& operator=(const basic_fixed_string& str)
  noexcept;
constexpr basic_fixed_string& operator=(const charT(&arr)[N + 1])
  noexcept;

constexpr operator view() const noexcept;

// iterators
constexpr iterator       begin() noexcept;
constexpr const_iterator begin() const noexcept;
constexpr iterator       end() noexcept;
constexpr const_iterator end() const noexcept;

constexpr reverse_iterator       rbegin() noexcept;
constexpr const_reverse_iterator rbegin() const noexcept;
constexpr reverse_iterator       rend() noexcept;
constexpr const_reverse_iterator rend() const noexcept;

constexpr const_iterator         cbegin() const noexcept;
constexpr const_iterator         cend() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;

// capacity
constexpr size_type size() const noexcept;
constexpr size_type length() const noexcept;
constexpr size_type max_size() const noexcept;
constexpr size_type capacity() const noexcept;
constexpr bool empty() const noexcept;

// XY.N.M+3, element access
constexpr const_reference operator[](size_type pos) const noexcept;
constexpr reference operator[](size_type pos) noexcept;
constexpr const_reference at(size_type pos) const noexcept;
```

```
constexpr reference at(size_type pos) noexcept;

constexpr const_reference front() const noexcept;
constexpr reference front() noexcept;
constexpr const_reference back() const noexcept;
constexpr reference back() noexcept;

// modifications
constexpr basic_fixed_string& replace(size_t pos, view str);
constexpr void swap(basic_fixed_string& str);

// string operations
constexpr const charT* c_str() const noexcept;
constexpr const charT* data() const noexcept;

constexpr size_type find(view str, 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 find(charT c, size_type pos = 0) const noexcept;
constexpr size_type rfind(view str,
                          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 rfind(charT c, size_type pos = npos) const;

constexpr size_type find_first_of(view str,
                                  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_first_of(charT c, size_type pos = 0) const;
constexpr size_type find_last_of(view str,
                                 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;
constexpr size_type find_last_of(charT c, size_type pos = npos) const;
constexpr size_type find_first_not_of(view str, size_type pos = 0)
  const noexcept;
constexpr size_type find_first_not_of(const charT* s, size_type pos,
                                      size_type n) const;
constexpr size_type find_first_not_of(const charT* s,
                                      size_type pos = 0) const;
constexpr size_type find_first_not_of(charT c, size_type pos = 0) const;
```

```
    constexpr size_type find_last_not_of(view str, size_type pos = npos)
      const noexcept;
    constexpr size_type find_last_not_of(const charT* s, size_type pos,
                                         size_type n) const;
    constexpr size_type find_last_not_of(const charT* s,
                                         size_type pos = npos) const;
    constexpr size_type find_last_not_of(charT c,
                                         size_type pos = npos) const;

    template <size_type pos = 0, size_type count = npos>
      constexpr basic_fixed_string<charT, /*...*/> substr() const noexcept;
    constexpr int compare(view str) const noexcept;
    constexpr int compare(size_type pos1, size_type n1, view str) const;
    constexpr int compare(size_type pos1, size_type n1, view str,
                          size_type pos2, size_type n2 = npos) 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;

  private:
    charT data_[N + 1];  // exposition only
                         // (+1 is for terminating null)
  };

}  // namespace fundamentals_vXXXX
}  // namespace experimental
}  // namespace std
```