Compile-Time String: std::string_literal<n>

Document Number: N4121
Project: Programming Languages C++, Library Evolution Working Group
Reply-To: Andrew Tomazos <andrewtomazos@gmail.com>
Date: 2014-07-04

Summary

We propose the addition of a class template `std::string_literal<n>` to the LFTS (and eventually merged into the Strings Standard Library) for the purposes of working with text at compile-time. The proposed template is a pure library helper class and does not require core language changes or compiler-support.

This proposal is depended upon and is blocking N3815 -> N4027 -> N4113 (Reflection Type Traits) from SG7, which needs to produce compile-time text for reflection of entity names.

High-Level Design

The proposed template fills a conceptual gap in the standard library which can be seen as follows:

std::vector	std::array	built-in array	std::array_view
std::string	std::string_literal	built-in string literal	std::string_view

Table 1

The purpose of std::string_literal, like std::string, is to provide a convenience utility for working with text. Unlike std::string, an instantiation of std::string_literal is a literal type and so can be used at compile-time. That is, it may be the type of an constexpr object, and it may be the type of a parameter, return value or local variable of a constexpr function.

This is achieved (like the built-in string literal) by making the number of code units part of the type. This is so that the storage requirements are fixed, and the code units can be stored directly as subobjects of the std::string_literal, rather than allocated with dynamic storage duration, which is not possible during compilation.

Alternative Designs

What about just making std::string a literal type?

This would require a massive core language change to make something like dynamic memory available at compile-time, or to make something like VLA/ARB and permit them in literal types. Given the violently negative reaction of Rapperswil Evolution to not only N4025 (Classes of Runtime Size), but anything that vaguely resembles VLA/ARBs, we can expect this not to happen any time soon, so this idea is a non-starter.

What about making the code units themselves a non-type template parameter pack like in N3933?

The design decision here is roughly between:

```
template<size_t n> struct string_literal { char data[n]; }
```

and

```
template<char... data> struct string_literal {}
```

First, to properly frame the discussion, we would like to demonstrate that you can form a `std::string_literal<n>` from such a char pack, and then we will show how to do the opposite, convert a `std::string_literal<n>` to a char pack.

Suppose we are have such a pack in scope:

```
template<char... input>
void f()
```

then we can just expand it into an initializer list...

```
{
    constexpr std::string_literal<sizeof...(input)> output = {input...};
```

```
};
```

Ok, that was easy.

Now the reverse, suppose we have an input `std::string_literal<n>`:

```
constexpr std::string_literal<n> input = /* ... */;
```

And we have an output template that needs a char pack:

```
template<char...> struct output_template;
```

So we just need to:

```
template<const std::string_literal<n>& s, typename T> struct x;
template<const std::string_literal<n>& s, size_t... i>
struct x<s, std::index_sequence<i...>>
{
    using type = output_template<s[i]...>;
};

using output = x<input, std::make_index_sequence<input.size()>>::type;
```

Ok, non-trivial, but still only 7 lines of standard C++. We also expect future core features in C++17 and beyond that will make working with packs easier (for example a core language replacement for std::index_sequence)

With that out of the way, our impression after studying the pros/cons is that having a different instantiation (different type) for every string literal per-content it contains isn't scalable and puts undue pressue on the implementation and linker as it has to mangle and merge this potentially extremely large set of types. Passing around text in char packs, as a matter of course, seems to be an abuse of not only the intended design of both non-type template parameters and variadic templates, but at a practical level the way implementations handle them (at least currently) seems to cause concrete performance problems.

At a higher-level, as shown in Table 1, the orthogonality of the relationships between the eight entities seems to more naturally suggest just using the size (as per std::array, built-in array and the type of a built-in string literal), and leave the content where it belongs, as part of the value of the type. Notice that in none of the 7 existing entities in Table 1 is the content part of the type.

From yet another point-of-view, we can see `std::string_literal<N>` as the C++14 constexpr-style version, and the `std::string_literal<cs...>` as the older template metaprogramming style version. In general, we have found that people prefer to use the C++14 constexpr-style for compile-time programming as it is closer to regular runtime C++

programming, and was designed from the outset for compile-time programming, rather than it being an accidental after-thought like when template metaprogramming was found to be turing complete and template instantiation “just happened” to work like functional programming.

Finally the minimum number of template arguments allowed by an implementation is 1024, which means that an implementation may only support char packs containing only 1024 code units. Arrays on the other hand have much higher limits.

We understand that the distinction is highly subjective and arguable, and are happy to have this design decision discussed and put to a vote.

Interface Design Process

Our interface design process is to start with a merged superset of the interfaces of `std::string` and `std::string_view`, and then remove everything that cannot be implemented with a fixed size or that would disqualify `std::string_literal` as a literal type.

Draft Synopsis <string_literal>

There is still a lot of fine tuning to be done, and everything needs to be tested for feasibility and correctness, and we need to think about integration with std::string and std::string_view (all the conversions), and the interface delta from std::string -> std::string_view has not yet been integrated, but we wanted to share this early draft of the interface to give an overview of the approach for feedback. The functions all behave the same as std::string:

```
template<class charT, size_t N, class traits = char_traits<charT> >
class basic_string_literal;

// std::make_string_literal("foo");
template<class charT, size_t N, class traits = char_traits<charT> >
constexpr basic_string_literal<charT, N, traits>
    make_string_literal(const charT(&arr)[N]);

// concatenation:  s1 + s2; s1 + "foo"; s1 + 'c';
template<class charT, class traits, size_t N, size_t M>
constexpr basic_string_literal<charT, N+M, traits>
operator+(const basic_string_literal<charT, N, traits>& lhs,
     const basic_string_literal<charT, M, traits>& rhs) noexcept;

template<class charT, class traits, size_t N1, size_t M>
constexpr basic_string_literal<charT, N1-1+M, traits>
operator+(const charT(&lhs)[N1],
```

```
      const basic_string_literal<charT, M, traits>& rhs) noexcept;

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

template<class charT, class traits, size_t N, size_t M1>
constexpr basic_string_literal<charT, N+M1-1, traits>
operator+(const basic_string_literal<charT, N, traits>& lhs,
          const charT (&rhs)[M1]) noexcept;

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

// comparison: s1 == s2; s1 < s2; s1 == "foo"; s1 < "foo"
template<class charT, class traits, size_t N, size_t M>
constexpr bool operator==(
      const basic_string_literal<charT, N, traits>& lhs,
      const basic_string_literal<charT, M, traits>& rhs)
      noexcept;

template<class charT, class traits, size_t N1, size_t M>
constexpr bool operator==(const charT (&lhs)[N1],
     const basic_string_literal<charT,M,traits>& rhs)
     noexcept;

template<class charT, class traits, size_t N, size_t M1>
constexpr bool operator==(
     const basic_string_literal<charT,N,traits>& lhs,
     const charT (&rhs)[M1]) noexcept;

template<class charT, class traits, size_t N, size_t M>
constexpr bool operator!=(
     const basic_string_literal<charT, N, traits>& lhs,
     const basic_string_literal<charT, M, traits>& rhs)
     noexcept;

template<class charT, class traits, size_t N1, size_t M>
constexpr bool operator!=(const charT (&lhs)[N1],
     const basic_string_literal<charT, M, traits>& rhs)
```

```
     noexcept;

template<class charT, class traits, size_t N, size_t M1>
constexpr bool operator!=(
     const basic_string_literal<charT,N,traits>& lhs,
     const charT (&rhs)[M1]) noexcept;

template<class charT, class traits, size_t N, size_t M>
constexpr bool operator<(
     const basic_string_literal<charT, N, traits>& lhs,
     const basic_string_literal<charT, M, traits>& rhs)
    noexcept;

template<class charT, class traits, size_t N1, size_t M>
constexpr bool operator<(const charT (&lhs)[N1],
    const basic_string_literal<charT, M, traits>& rhs)
    noexcept;

template<class charT, class traits, size_t N, size_t M1>
constexpr bool operator<(
     const basic_string_literal<charT, N, traits>& lhs,
     const charT (&rhs)[M1]) noexcept;

template<class charT, class traits, size_t N, size_t M>
constexpr bool operator>(
     const basic_string_literal<charT, N, traits>& lhs,
     const basic_string_literal<charT, M, traits>& rhs)
    noexcept;

template<class charT, class traits, size_t N1, size_t M>
constexpr bool operator>(const charT (&lhs)[N1],
    const basic_string_literal<charT, M, traits>& rhs)
    noexcept;

template<class charT, class traits, size_t N, size_t M1>
constexpr bool operator>(
     const basic_string_literal<charT, N, traits>& lhs,
     const charT (&rhs)[M1]) noexcept;

template<class charT, class traits, size_t N, size_t M>
constexpr bool operator<=(
     const basic_string_literal<charT, N, traits>& lhs,
     const basic_string_literal<charT, M, traits>& rhs)
```

```
      noexcept;

template<class charT, class traits, size_t N1, size_t M>
constexpr bool operator<=(const charT (&lhs)[N1],
      const basic_string_literal<charT, M, traits>& rhs)
      noexcept;

template<class charT, class traits, size_t N, size_t M1>
constexpr bool operator<=(
      const basic_string_literal<charT, N, traits>& lhs,
      const charT (&rhs)[M1]) noexcept;

template<class charT, class traits, size_t N, size_t M>
constexpr bool operator>=(
      const basic_string_literal<charT, N, traits>& lhs,
      const basic_string_literal<charT, M, traits>& rhs)
      noexcept;

template<class charT, class traits, size_t N1, size_t M>
constexpr bool operator>=(const charT (&lhs)[N1],
      const basic_string_literal<charT, M, traits>& rhs)
      noexcept;

template<class charT, class traits, size_t N, size_t M1>
constexpr bool operator>=(
      const basic_string_literal<charT, N, traits>& lhs,
      const charT (&rhs)[M1]) noexcept;

// swap(s1,s2)
template<class charT, class traits, size_t N>
constexpr void swap(
      basic_string_literal<charT, N, traits>& lhs,
      basic_string_literal<charT, N, traits>& rhs) noexcept;

// std::cout << s1;
template<class charT, class traits, size_t N>
    basic_ostream<charT, traits>& operator<<(
    basic_ostream<charT, traits>& os,
    const basic_string_literal<charT, N, traits>& str);

// string_literal<N>
template<size_t N> using string_literal
      = basic_string_literal<char, N>;
```

```
template<size_t N> using u16string_literal
     = basic_string_literal<char16_t, N>;
template<size_t N> using u32string_literal
     = basic_string_literal<char32_t, N>;
template<size_t N> using wstring_literal
     = basic_string_literal<wchar_t, N>;

// constexpr int x = stoi(s1);
template<size_t N>
constexpr int stoi(const string_literal<N>& str,
                   size_t* idx = 0, int base = 10);
template<size_t N>
constexpr long stol(const string_literal<N>& str,
                    size_t* idx = 0, int base = 10);
template<size_t N>
constexpr unsigned long stoul(const string_literal<N>& str,
                          size_t* idx = 0, int base = 10);
template<size_t N>
constexpr long long stoll(const string_literal<N>& str,
                          size_t* idx = 0, int base = 10);
template<size_t N>
constexpr unsigned long long stoull(
     const string_literal<N>& str,
     size_t* idx = 0, int base = 10);

template<size_t N>
constexpr float stof(
     const string_literal& str, size_t* idx = 0);
template<size_t N>
constexpr double stod(
     const string_literal& str, size_t* idx = 0);
template<size_t N>
constexpr long double stold(
     const string& str, size_t* idx = 0);

// constexpr auto s1 = to_string_literal_i<42>; // = "42"
template<int val>
constexpr string_literal</*impl-defn*/>
    to_string_literal_i;

template<unsigned val>
constexpr string_literal</*impl-defn*/>
    to_string_literal_u;
```

```
template<long val>
constexpr string_literal</*impl-defn*/>
    to_string_literal_l;

template<unsigned long val>
constexpr string_literal</*impl-defn*/>
    to_string_literal_ul;

template<long long val>
constexpr string_literal</*impl-defn*/>
    to_string_literal_ll;

template<unsigned long long val>
constexpr string_literal</*impl-defn*/>
    to_string_literal_ull;

// same for wstring_literal<N>
template<size_t N>
constexpr int stoi(const wstring_literal<N>& str,
                   size_t* idx = 0, int base = 10);
template<size_t N>
constexpr long stol(const wstring_literal<N>& str,
                    size_t* idx = 0, int base = 10);
template<size_t N>
constexpr unsigned long stoul(
     const wstring_literal<N>& str,
     size_t* idx = 0, int base = 10);

template<size_t N>
constexpr long long stoll(
     const wstring_literal<N>& str, size_t* idx = 0,
     int base = 10);

template<size_t N>
constexpr unsigned long long stoull(
     const wstring_literal<N>& str,
        size_t* idx = 0, int base = 10);

template<size_t N>
constexpr float stof(
         const wstring_literal<N>& str, size_t* idx = 0);
```

```
template<size_t N>
constexpr double stod(const wstring_literal<N>& str,
                      size_t* idx = 0);

template<size_t N>
constexpr long double stold(
     const wstring_literal<N>& str,
    size_t* idx = 0);

template<int val>
constexpr wstring_literal</*impl-defn*/>
    to_wstring_literal_i;

template<unsigned val>
constexpr wstring_literal</*impl-defn*/>
    to_wstring_literal_u;

template<long val>
constexpr wstring_literal</*impl-defn*/>
    to_wstring_literal_l;

template<unsigned long val>
constexpr wstring_literal</*impl-defn*/>
    to_wstring_literal_ul;

template<long long val>
constexpr wstring_literal</*impl-defn*/>
    to_wstring_literal_ll;

template<unsigned long long val>
constexpr wstring_literal</*impl-defn*/>
    to_wstring_literal_ull;

// user defined literal: auto s1 = "foo"sl
// (depends on N3599)
template<typename charT, charT... cs>
constexpr basic_string_literal<charT, sizeof...(cs)>
operator""sl ();

// basic_string_literal definition
template<class charT, size_t N, class traits
     = char_traits<charT>>
class basic_string_literal
```

```
{
public:
      typedef traits traits_type;
      typedef typename traits::char_type value_type;

      typedef value_type& reference;
      typedef const value_type& const_reference;
      typedef value_type* pointer;
      typedef const value_type* const_pointer;

      typedef pointer iterator;
      typedef const_pointer const_iterator;
      typedef std::reverse_iterator<iterator>
          reverse_iterator;
      typedef std::reverse_iterator<const_iterator>
          const_reverse_iterator;
      static constexpr size_type npos = -1;

      constexpr basic_string_literal() noexcept;
      constexpr basic_string_literal
          (const basic_string_literal& str) noexcept;

      ~basic_string();

      constexpr basic_string& operator=
           (const basic_string_literal& str) noexcept;
      constexpr basic_string& operator=
           (const charT (&s)[N+1]);
      constexpr basic_string& operator=
           (charT c);
      constexpr basic_string& operator=
           (initializer_list<charT>);

      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;

constexpr size_t size() const noexcept;
constexpr size_t length() const noexcept;

constexpr bool empty() const noexcept;

constexpr const_reference operator[](size_t pos)
    const noexcept;

constexpr reference operator[](size_t pos) noexcept;
constexpr const_reference at(size_type n) const;
constexpr reference at(size_type n);

constexpr const charT& front() const;
constexpr charT& front();
constexpr const charT& back() const;
constexpr charT& back();

constexpr basic_string_literal& assign(
     const basic_string_literal& str);
constexpr basic_string_literal& assign(
     const charT(&s)[N]);

constexpr template<class InputIterator>
basic_string& assign(InputIterator first,
                     InputIterator last);
    // last - first must == N

constexpr basic_string_literal& assign(
     initializer_list<charT>);
    // initializer_list length must == N

template <size_t M>
constexpr basic_string_literal& replace(size_type pos1,
```

```
    size_type n1,
    const basic_string_literal<charT, M, traits>& str);

template <size_t M>
constexpr basic_string_literal& replace(
     size_type pos1, size_type n1,
     const basic_string_literal<charT,M,traits>& str,
     size_type pos2, size_type n2 = npos);

constexpr basic_string_literal& replace(
     size_t pos, size_t n1,
     const charT* s, size_type n2);

constexpr basic_string_literal& replace(size_t pos,
    size_t n1, const charT* s);

constexpr basic_string_literal& replace(size_t pos,
    size_t n1, size_t n2, charT c);

template <size_t M>
constexpr basic_string_literal& replace(
     const_iterator i1, const_iterator i2,
     const basic_string_literal<charT, M, traits>& str);

constexpr basic_string_literal& replace(
     const_iterator i1, const_iterator i2,
     const charT* s, size_type n);

constexpr basic_string_literal& replace(
     const_iterator i1, const_iterator i2,
     const charT* s);

constexpr basic_string_literal& replace(
     const_iterator i1, const_iterator i2,
     size_t n, charT c);

template<class InputIterator>
constexpr basic_string_literal& replace(
     const_iterator i1, const_iterator i2,
     InputIterator j1, InputIterator j2);

constexpr basic_string& replace(
     const_iterator, const_iterator,
```

```
      initializer_list<charT>);

constexpr size_type copy(charT* s,
    size_type n, size_type pos = 0) const;

constexpr void swap(basic_string_literal& str);

constexpr const charT* c_str() const noexcept;
constexpr const charT* data() const noexcept;
constexpr allocator_type get_allocator()
    const noexcept;

template<size_t M>
constexpr size_t find (
      const basic_string_literal<charT,M,traits>& str,
      size_t pos = 0) const noexcept;

constexpr size_t find (
      const charT* s, size_t pos, size_t n) const;

constexpr size_t find (
      const charT* s, size_t pos = 0) const;

constexpr size_t find (charT c, size_t pos = 0) const;

template<size_t M>
constexpr size_t rfind(
      const basic_string_literal<charT,M,traits>& str,
      size_t pos = npos) const noexcept;

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

constexpr size_t rfind(
      const charT* s, size_t pos = npos) const;

constexpr size_t rfind(
      charT c, size_t pos = npos) const;

template<size_t M>
constexpr size_t find_first_of(
      const basic_string_literal<charT,M,traits>& str,
      size_t pos = 0) const noexcept;
```

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

constexpr size_t find_first_of(
     const charT* s, size_t pos = 0) const;

constexpr size_t find_first_of(
     charT c, size_type pos = 0) const;

template<size_t M>
constexpr size_t find_last_of (
     const basic_string_literal<charT,M,traits>& str,
     size_t pos = npos) const noexcept;

constexpr size_t find_last_of (
     const charT* s, size_t pos, size_t n) const;

constexpr size_t find_last_of (
     const charT* s, size_t pos = npos) const;

constexpr size_t find_last_of (
     charT c, size_t pos = npos) const;

template<size_t M>
constexpr size_t find_first_not_of(
     const basic_string_literal<charT,M,traits>& str,
     size_t pos = 0) const noexcept;

constexpr size_t find_first_not_of(
     const charT* s, size_t pos, size_t n) const;

constexpr size_t find_first_not_of(const charT* s,
    size_t pos = 0) const;
constexpr size_t find_first_not_of(
     charT c, size_t pos = 0) const;

template<size_t M>
constexpr size_t find_last_not_of (
     const basic_string_literal<charT,M,traits>& str,
     size_t pos = npos) const noexcept;

constexpr size_t find_last_not_of (
```

```
        const charT* s, size_t pos, size_t n) const;

    constexpr size_t find_last_not_of (
        const charT* s, size_t pos = npos) const;
    constexpr size_t find_last_not_of (
        charT c, size_t pos = npos) const;

    template<size_t M> constexpr
    basic_string_literal<charT, M, traits>
    substr(size_t pos = 0) const;

    template<size_t M> constexpr int compare(
        const basic_string_literal<charT, M, traits>& str)
      const noexcept;

    template<size_t M> constexpr int compare(
        size_t pos1, size_t n1,
        const basic_string_literal<charT, M, traits>& str)
      const;

    template<size_t M> constexpr int compare(
        size_t pos1, size_t n1,
        const basic_string_literal<charT, M, traits>& str,
        size_type pos2, size_type n2 = npos) const;

    constexpr int compare(const charT* s) const;
    constexpr int compare(size_t pos1, size_t n1,
                          const charT* s) const;
    constexpr int compare(size_t pos1, size_t n1,
                          const charT* s, size_t n2) const;

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