Doc. no.: P0903R0
Date: 2018-02-02
Reply to: Ashley Hedberg (ahedberg@google.com),
Audience: LEWG/LWG

Define `basic_string_view(nullptr)` and `basic_string(nullptr)`

Abstract

This paper proposes defining `char_traits<T>::length(s)` for `s == nullptr` and modifying the requirements of `basic_string(const charT* s, const Allocator& a = Allocator())` such that `basic_string_view(const charT* str)` and `basic_string(const charT* s, const Allocator& a = Allocator())` become well-defined for null pointers.

Background

Current behavior of `string_view` constructors

`basic_string_view(nullptr)` is currently undefined behavior. Such code invokes the `basic_string_view(const charT* str)` constructor, which requires that `[str, str + traits::length(str))` is a valid range [string.view.cons]. The current wording on requirements for `char_traits<T>::length` is as follows [char.traits.require]:

> *Returns*: the smallest `i` such that `X::eq(p[i], charT())` is `true`.

There is no such `i` when `p` is null. Thus, `basic_string_view(nullptr)` is undefined.

Conversely, `basic_string_view()` and `basic_string_view(nullptr, 0)` are both defined to construct an object with `size_ == 0` and `data_ == nullptr` [string.view.cons].

Current behavior of `string` constructors

`basic_string(nullptr)` is currently undefined behavior. Such code invokes the `basic_string(const charT* s, const Allocator& a = Allocator())` constructor, which requires that `s` points to an array of at least `traits::length(s) + 1` elements of `charT` [string.cons]. As described above, `traits::length(s)` is undefined when `s` is null. Thus, `basic_string(nullptr)` is undefined.

Conversely, `basic_string()` and `basic_string(nullptr, 0)` are both defined to construct an object with `size() == 0` [string.cons].

Motivation

Motivation for defining `string_view(nullptr)`

Having a well-defined `basic_string_view(nullptr)` makes migrating `char*` APIs to `string_view` APIs easier. Here's an example API which we may wish to migrate to `string_view`:

```
void foo(const char* p) {
  if (p == nullptr) return;
  // Process p
}
```

Callers of `foo` can pass null or non-null pointers without worry. However, this function cannot be safely migrated to accept `string_view` unless one can **statically** determine that no null `char*` is ever passed to it:

```
void foo(std::string_view sv) {
  if (sv.empty()) return;  // Too late - constructing sv from nullptr is undefined!
  // Process sv
}
```

If `basic_string_view(nullptr)` becomes well-defined, APIs currently accepting `char*` or `const string&` can all move to `std::string_view` without worrying about whether parameters could ever be null.

This change also makes instantiating empty `string_view` objects more consistent across constructors. `basic_string_view()`, `basic_string_view(nullptr)`, and `basic_string_view(nullptr, 0)` will all construct an object with `size_ == 0` and `data_ == nullptr`. Furthermore, it increases consistency across library versions without penalty. libstdc++, the proposed `std::span`, `absl::string_view`, and `gsl::string_span` already support constructing a `string_view`-like object from a null pointer with no size; libc++ and MSVC do not.

Motivation for defining `string(nullptr)`

With the above proposal, `basic_string_view()`, `basic_string_view(nullptr)`, `basic_string_view(nullptr, 0)`, `basic_string()`, and `basic_string(nullptr, 0)` would all be well-defined. Defining `basic_string(nullptr)` makes instantiating empty `string` objects more consistent across constructors of that class, and is consistent with the proposed behavior for `string_view`.

libstdc++ already supports constructing a `string` object from a null pointer with no size; libc++ and MSVC do not.

Proposed Wording

Define `char_traits<T>::length` for null arguments

Change the Assertion/note pre-/post-condition column for the expression `X::length(p)` as follows [char.traits.require]:

> *Returns*: 0 if `p == nullptr`; else, the smallest `i` such that `X::eq(p[i],charT())` is `true`.

Changes to `basic_string(const charT* s, const Allocator& a = Allocator())`

Change the requirements for `basic_string(const charT* s, const Allocator& a = Allocator())` as follows [string.cons]:

> *Requires*: if `s != nullptr`, `s` points to an array of at least `traits::length(s) + 1` elements of `charT`

Considerations

The proposed `char_traits<T>::length` change would cause both `traits::length(nullptr)` and `traits::length("")` to return 0. This is ambiguous. However, `basic_string_view("")` and `basic_string_view(nullptr, 0)` both construct objects where `size() == 0`, so there is precedent for this ambiguity.

The proposed `char_traits<T>::length` change also requires its implementations to check for `nullptr` and branch accordingly. However, `char_traits<T>::length` is already an O(n) operation in the non-null case, so the cost of a branch is much smaller relative to the existing behavior.

Alternative Wordings

If inserting a branch in `char_traits<T>::length` is undesirable, the `basic_string_view(const charT* str)` constructor could be changed instead:

Change the requirements and effects for `basic_string_view(const charT* str)` as follows [string.view.cons]:
Requires: if `str != nullptr`, `[str, str + traits::length(str))` is a valid range.
Effects: Constructs a `basic_string_view`, with the postconditions in Table 56:

Table 56 -- `basic_string_view(const charT*)` effects

Element	**Value**
`data_`	`str`
`size_`	0 if `str == nullptr`; else, `traits::length(str)`

Acknowledgements

- Titus Winters for proposing that I write this proposal.
- Matt Calabrese for assistance in navigating existing committee papers, notes. etc.
- Titus Winters, Matt Calabrese, John Olson for providing feedback on drafts of this proposal.