Doc. no.: P0903R2
Date: 2018-05-07
Reply to: Ashley Hedberg (ahedberg@google.com), Titus Winters (titus@google.com), Jorg Brown (jorg.brown@gmail.com)
Audience: LEWG/EWG/LWG

Define `basic_string_view(nullptr)`

Abstract

This paper proposes modifying the requirements of `basic_string_view(const charT* str)` such that it becomes well-defined for null pointers, both at compile-time and at runtime.

Background

What is a `string_view`?

The `string_view` type has shipped in C++17. Substantive redesign is likely out of the question without getting into P0684 territory and long-term refactoring plans.

There are two primary viewpoints for understanding the design of `string_view` as it stands. Both are internally inconsistent to varying degrees.

1. `string_view` is a lightweight non-owning string. This is upheld by looking at its `operator==`, which compares by examining the underlying string data--not pointer values. In this model, the fact that `string_view()` has post-conditions on `.data() == null` is questionable. Arguably, access to `.data()` at all is questionable in this view: it's a non-null-terminated `char*`, and is a side-channel/leaky abstraction on the type akin to `.capacity()` on a `vector`. It isn't part of the logical state as defined by `operator==`.
2. `string_view` is a pointer and a length over a buffer of string data that it does not own. In this model, `operator==` is questionable: it's generally the case that what an object *is* and what it compares should be the same. [That said, that rule was developed for Regular types, and `string_view` is decidedly not Regular (P0898).] The various constructors and assignment operations and accessors (`.data()`, `.size()`) are consistent in this model.

Fundamentally, `string_view` has a troubled design when evaluated under our understanding of Regular types. There is a mismatch between what is copied and what is compared. Which pieces of API you regard as intrinsic and which you regard as questionable depend entirely on which model you use to describe the type. (We believe that this may be primarily indicative of us not having a shared understanding of how to discuss non-owning types.)

One author of this paper has been running classes that touch on `string_view` for about six years. Part of that includes an attempt to give people a working mental model for `string_view` to avoid the (constant) issues of people messing up the relative lifetimes of `string_view` and the underlying buffer. As such, the terminology that the author uses (and requires the class to repeat out loud) is “a `string_view` is a pointer and a length to a buffer it does not own and cannot modify.”

What does `const char*` mean?

In almost all cases in the C and C++ standards today, a `const char*` indicates a non-null C-style (nul-terminated) string. `string_view` weakens this somewhat, even in the absence of this proposal: `string_view::data()` returns a not-necessarily nul-terminated `const char*`.

Outside of the C and C++ standards, there are several commonly-used use cases on the Linux platform where a `const char*` can be null. For example:

- `getenv` returns a `const char*` that might be null. Null indicates that there is no environment variable with that name; an empty `const char*` indicates that there is such an environment variable defined with the value "".
- `inet_ntop` returns null to indicate an error.
- `system` accepts a `const char*` parameter. If the parameter is null, `system` returns whether or not a shell is available.

It is not unusual in the face of such interfaces for programmers to work with const char*s that are null.

What is an empty `string_view`?

A `string_view` will report `empty() == true` so long as `.size() == 0`. That is, there are empty `string_view`s that have non-null `.data()`.

Put another way: a null `string_view` is a certain sub-configuration of the set of empty `string_view`s.

Code that operates on `string_view` which relies on those differences has been seen in the wild, but almost universally is difficult to work with. It isn't a recommended design, but it is a direct side-effect of the current design of the type. Such usage is generally leaning on the view of "`string_view` is a pointer and length" rather than "`string_view` is a lightweight non-owning string" (or at best is relying on known side-channels for `.data()` in the non-owning string conceptualization). The standard library is not the place to enforce good taste, and this is existing behavior that we can't fix, even if we don't like it.

Existing behavior of `basic_string_view(null_char_ptr)`

Throughout this paper, `null_char_ptr` is a null pointer of type `const char*` (e.g. `nullptr`, `NULL`, `0`).

`basic_string_view(null_char_ptr)` 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(null_char_ptr)` is undefined.

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

Motivation

Once we convert into the `string_view` domain, code is better

Having a well-defined `basic_string_view(null_char_ptr)` 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 null is undefined!
  // Process sv
}
```

If `basic_string_view(null_char_ptr)` 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. In legacy codebases with long chains of function calls, that question may not be easily determined.

This change also makes instantiating empty `string_view` objects more consistent across constructors. `basic_string_view()`, `basic_string_view(null_char_ptr)`, and `basic_string_view(null_char_ptr, 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.

Barring unsafe conversions (calling `string_view::data()` to get a C-style string), once a function is written in terms of `string_view`, it tends to be higher quality than the equivalent with `const char*`. For example:

Copies into string are explicit:

```
void AlreadyHasCharStar(const char* s) {
  TakesString(s);  // compiler will make a non-obvious copy
}

void AlreadyHasStringView(std::string_view sv) {
  TakesString(string(sv));  // copy is explicit
}
```

Operations on `string_view` use higher-level APIs such as `find`:

```
bool CharStarContains(
    const char* s, const char* sub) {
  return strstr(s, sub) != nullptr;
}

bool StringViewContains(
    std::string_view sv, std::string_view sub) {
  return sv.find(sub);
}
```

More generally: it's harder to misuse `string_view` than `const char*`. Encouraging broader and more consistent use of the higher-level type nudges us toward better code quality.

Nul-terminated strings are not the design we'd choose today

Why do we use nul-terminated strings? Where did that convention come from? Upon consideration, embedding the terminator is clearly a bit of a hack to avoid passing (pointer + length) pairs everywhere. This may have been a sensible choice when memory dereference mattered less and CPU and register pressure mattered more, but it's clearly not ideal. We're modifying the underlying buffer to encode the length, preventing `string_view`-like operations over arbitrary sub-ranges of text.

Types with a sentinel "invalid value" are annoying and hard to work with. Consider floating-point: we call `float` and `double` "regular" with big footnotes saying "ignoring NaN". Worse: it's a little difficult to get a `float` into a NaN state, but `const char*`'s most common state / zero-initialization state is its "invalid value" sentinel. This isn't the design we would produce today if we were thinking of how to represent string data. The design and semantics of `const char*` APIs are what we're stuck with, but they are not inherently good. If we have opportunity to mitigate that design or help the community get off of them *en masse*, it may be wise to do so.

Nul-termination + null can match string_view's existing design

There's a direct parallel to be drawn between `const char*` empty/null behavior and `string_view`. In `const char*`, the empty string is "", and a null is null. In `string_view`, the empty `string_view` (as reported by `.empty()`) is anything with `size() == 0`, and a null `string_view` is the subset of those where `.data()` also points to null (such as when default constructing). Again, we might not like code which relies on these distinctions--but such code already exists, and we can't easily undo either design decision at this point.

Discussion Points

After sifting through hundreds of mailing list messages on this topic, we believe that the major questions that inform one opinion or another are roughly as follows:

Can we we weaken preconditions? (Or: do we believe that `string_view(null_char_ptr)` is more likely to match programmer intent or to represent a bug?)

There is an often-quoted maxim when discussing refactoring: we can weaken the preconditions or strengthen the postconditions for a function safely. However, this is somewhat misleading: those changes are "safe" in the sense that for existing correct code, the code will continue to be correct. What is missing from that formulation is the creation of new code, and the understanding that newly-created code is often not correct from the start.

For instance, we could strengthen the postconditions for signed integer overflow: all instances of overflow produce the value 42. In existing correct code, this has no impact: there is no existing correct code that relies on overflow, by definition. However, in newly created code (or freshly modified code), the fact that overflow is no longer undefined doesn't mean that triggering this behavior is not a bug: it is unlikely that a programmer writing `int a = x + y;` *intends* for that to be "give me the sum of x and y unless it cannot be expressed in the range of an int, at which point give me the value 42."

With this example and model in hand we can see that undefined behavior is valuable at minimum in cases where we cannot correctly match the intent of the programmer. We can also see that there is a distinction to be made between UB and "bug" - some things may be defined but still buggy.

This leads us to the question: do we believe that the proposed behavior for `string_view(null_char_ptr)` is more likely to match programmer intent or to represent a bug? This question depends on many inputs, including (but not limited to):

- Whether you view `string_view` as a nullable type.
- Whether it is inherently bad for any `const char*` API to allow null.
- Whether you believe it is reasonable shorthand for a user to conflate any of the properties of "" and nullptr.

Similarly, is `string_view(vector.data(), vector.size())` a real or theoretical counter-argument to the existence of a null/not-a-string `string_view`? Even if it's a construction that is found in the wild, is that actually indicative of an issue in `string_view`, or in `vector`'s design? Or just user-error? The authors have been using `string_view` for many years and have never seen such a construction, much less a complaint about the confusing semantics as a result.

How many functions are there today that accept `const char*` and (by contract or not) allow for null values?

If there are many such functions: are they appropriate to migrate to `string_view`? If they tend to call OS or C-library APIs that require nul-terminators, migration is likely counter-productive.

Which view of `string_view` design speaks to you: is it a lightweight string, or is it a pointer-and-length pair, or is it a mix of the two?

If it's nullable or pointer-ish, it is less distasteful to assume 0 for length when handed null as the pointer. If it's a lightweight string with some leaky abstraction problems, the `const char*` really must be interpreted with standard C-string semantics.

Which provides more net value to the C++ community?

- The ability to change an API that accepts `const char*` to `string_view` with only local changes (header and implementation, with no modifications at the call site) AND with only standard types? Note that this does not imply that all such changes are mechanical or wise: if later functions in that call chain require nul-termination, it's often counter-productive to make such a change.
- The ability to diagnose (dynamically) future incorrect code more easily/consistently. Also, more complete consistency in the C++ and C standards with respect to the semantics of `const char*`-accepting APIs.

Could we add a new type such as `nullable_string_view`?

One could imagine implementing the following:

```cpp
struct nullable_string_view : public std::string_view {
  using std::string_view::string_view;

  constexpr nullable_string_view(const std::string_view& sv)
    : std::string_view(sv) {}
  constexpr nullable_string_view(const char* p)
      : std::string_view(p ? std::string_view(p) : std::string_view()) {}
};
```

Adding this type to the standard library is likely to confuse users and very likely a non-starter. Even adding it locally as a non-standard type raises a host of questions: When should one use `nullable_string_view` over `string_view`? Should a new API accept `nullable_string_view` to indicate that null `string_view`'s are tolerated, or should it accept `string_view` for wider interoperability? How would one handle a switch to a standard library that chooses to define this UB? (The two types must continue to exist and be distinct if they appear in any overload sets or template specializations.)

There's also the (not-specific to `string_view`) cost of introducing new overlapping vocabulary types. Which ones are taught? Which ones are included in overloads or template customizations? How do we provide best practices on the conversions between them and (perhaps more importantly) between them and other types?

Vocabulary types are conceptually expensive and can use up a lot of the available technical debt in any project. This approach is technically feasible, but not recommended. The perceived value from either approach to this constructor is less than the long-term cost of dealing with two overlapping types like this.

Will the proposed changes negatively affect performance?

The paper authors think not. This paper proposes adding a single check in an O(n) function that is already making more expensive checks. Deref-and-compare-to-zero is at least as expensive as compare-ptr-to-null. Furthermore, implementers stated in Jacksonville that the compiler should be able to elide the null pointer check when `null_char_ptr` is known to be `nullptr` at compile-time.

Proposed Wording

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* str)` effects

Element	**Value**
`data_`	`str`

`size_`	0 if `str == nullptr`; else `traits::length(str)`

Change History

R2 makes the following changes as a result of reflector discussion:

- Adds wording to the background section
- Adds wording to the motivation section
- Adds a "discussion points" section containing responses to arguments raised in reflector discussion

R1 makes the following changes as a result of LEWG feedback in Jacksonville:

- Removes suggested changes to `basic_string`.
- Makes the previous "alternate wording" the "proposed wording".
- Adds clarifying wording that the proposed change affects dynamically null pointers as well as statically null pointers.

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, Jorg Brown for providing feedback on drafts of this proposal.