P3710R0

`zstring_view`: a `string_view` with guaranteed null termination

Date: 2025-02-17
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: SG23, LEWG
Author: Alexey Shevlyakov, Marco Foco
Contributors: Joshua Krieghauser
Reply to: marco.foco@gmail.com

History

R0

Document creation

Overview

`zstring_view` is a lightweight, non-owning string reference which, unlike `std::string_view`, guarantees null-termination and is designed for interoperability with C-style strings and APIs that depend on a terminating null character.
To keep the exposition simple, throughout the document we'll discuss about the class `zstring_view` and the type `char*`, but without loss of generality, all the reasoning can be extended to a generic `basic_zstring_view<CharT, Traits>` class and to `CharT*` strings.

Motivation

Modern guidelines discourage using raw arrays for safety concerns, suggesting `std::string_view` as a replacement where a non-owning reference to a string is required. However, modern APIs exist alongside legacy APIs that cannot be changed for various reasons, leading to contradicting requirements and guarantees. Consider the following example:

```
// third party API that cannot be altered
extern void ext_foo(const char* str);
```

```
// legacy interface relying on null-terminated strings
void foo(const char* str){
  ext_foo(str);
}
```

We could replace the signature of foo:

```
void foo(std::string_view str){
  // LOGIC ERROR: ext_foo might rely on null-termination
  ext_foo(str.data());
}
```

However, we do not have control of `foo` because it's either a third-party API or a legacy API that we can't change for some other reason. Therefore, replacing the signature of foo with `string_view` is impossible, since consume relies on null-termination. Enter `zstring_view`. `zstring_view` is a `StringViewLike` object that guarantees that its content points to a null-terminated string and is interoperable with both null-terminated strings and `string_views`. It also maintains a subset of suffix-preserving member functions while lacking operations that do not preserve the suffix—e.g., `remove_prefix` is fine since it preserves the suffix and the null-terminator, while `remove_suffix` is not supported.

In addition, our class contains a new member function `c_str()`, which is used to retrieve the null-terminated string. This mimics the same behavior in `std::string::c_str()`, where the content is also guaranteed to be null-terminated.

Our example code can be rewritten using `zstring_view` as follows:

```
void foo(zstring_view str) {
  ext_foo(str.c_str()); // fine: str is null-terminated
}
```

`zstring_view` can also seamlessly interoperate with interfaces accepting `string_view`, as it represents a subset of `string_view` and is implicitly convertible to it (i.e. it is a `StringViewLike`):

```
extern void ext_bar(std::string_view str);

void bar(zstring_view str){
    ext_bar(str); // OK, zstring_view is convertible to string_view
}
```

Thus we are able to provide a safe and modern interface on the one hand and ensure its interoperability with legacy interfaces requiring C-style strings.

Constructors

`zstring_view` constructors are analogous to those of `string_view` whenever there is a possibility to check whether a null terminator is present at the right position.

Below we are going to highlight the differences between the constructors of `string_view` and `zstring_view`.
The default constructor initialises the data pointer to empty null-terminated string referencing a static member defined in the class as

```
static constexpr char empty_string[] = {char{}}

constexpr zstring_view() noexcept
  : m_data(empty_string), m_count(0) {}
```

The following constructor is for a pointer with length. Unlike the corresponding `string_view` constructor, it requires `s` to be a pointer to an array of char of size `count+1`, and a null terminator must be present at the end of the string (otherwise a precondition violation will happen).

```
constexpr zstring_view(const char* s, size_type count)
  pre(traits_type::eq(s[count], char{})
  : m_data(s), m_count(count)
  {}
```

In accordance with the `string_view` constructor from a bounded array, this constructor looks for the null terminator inside the array, and expects at least one to be present: the first null-terminator will be used to calculate the length of the string. If no terminator is present in the array, a precondition violation will happen.

```
template <size_t N>
constexpr zstring_view(const char (&s)[N])
  pre(traits_type::find(s, n+1, char{}) != nullptr)
  : m_data(s), m_length(traits_type::length_s(s))
{}
```

The "unsafe" constructor for a pointer expects a null-terminator to be present, and doesn't perform additional checks:

```
constexpr zstring_view(unsafe_length_t, const char* s) noexcept
  : m_data(s), m_count(traits_type::length(s))
```

```
{}
```

This constructor operates on anything that is similar to a `std::string` (i.e. has a `c_str()` and a `length()` member function) because it guarantees the presence of a null-terminator.

```
template <ZStringViewLike T>
constexpr string_view(const T& s)
  : m_data(s.c_str()), m_count(s.length())
{}
```

Unlike `string_view`, the range constructor is not supported by `zstring_view` because an arbitrary range doesn't guarantee the presence of a null-terminator.

Member functions

Most member functions of zstring_view are equivalent to those of string_view except for the functions that don't preserve the suffix.

All the following member functions have the same semantics as `string_view`: `operator=`, `begin`, `cbegin`, `end`, `cend`, `rbegin`, `crbegin`, `rend`, `crend`, `operator[]`, `at`, `front`, `back`, `size`, `length`, `max_size`, `empty`, `copy`, `compare`, `starts_with`, `ends_with`, `contains`, `find`, `rfind`, `find_first_of`, `find_last_of`, `find_first_not_of`, `find_last_not_of`.

Operations that don't preserve the suffix are not supported in `zstring_view`, so `remove_prefix` can be applied, but `remove_suffix` can't (the object would contain a non-terminated string).
For `substr`, we have two versions: one returning the tail of the string, which will return a `zstring_view` because the buffer will be properly null terminated:

```
constexpr zstring_view substr(size_t pos) const
  pre(pos =< size())
{
  return { c_str() + pos, size() - pos };
}
```

the other, with two parameters, cannot guarantee the presence of the null-termination within the substring, so it will simply return a `string_view`:

```
constexpr string_view substr(size_t pos, size_t count) const
  pre(pos =< size())
{
  return { c_str() + pos, std::min(count, size() - pos) };
```

```
}
```

Concepts

Similarly to P3566R1 we introduce a set of concepts that will allow us to build

- `ZStringViewLike`
  - Same as `StringViewLike`, but for `zstring_view`
  - Accepts `char*`
- `SafeZStringViewLike`
  - Similar to `StringViewLike`, but for `zstring_view` (i.e. `SafeZStringViewLike<T>` is true for all types `T` that are implicitly convertible to `zstring_view` as described in this paper: doesn't accept `char*`)
- `UnsafeZStringViewLike`
  - `ZStringViewLike && !SafeZStringViewLike`
  - `UnsafeZStringViewLike<T>` is true for all types `T` that are implicitly convertible to `zstring_view`, but are not `SafeZStringViewLike`

Usage Experience

We implemented `zstring_view` during a safety-improvement initiativethroughout our codebase, where we wanted to remove all the unsafe usages of C string functions (e.g. `strlen`, `strcpy`, `strdup`...) and migrate towards safer string constructs.

Other proposals we submitted to the committee, such as [P3566] and [P3711], are part of this journey, and reflect our experience in dealing with string safety issues.

We're currently using `zstring_view` as part of a migration path from `const char*` to `string_view`.
We asked our developers to be intentional when using `c_str()` vs `data()` member functions, respectively, to declare whether or not they expect the string to be null-terminator.

Introducing `zstring_view` we proposed these rule, which proved to be very effective: use `data()` when the null-terminator is not expected (e.g. because the pointer is passed to an external function without its length), and use `c_str()` when the code expects a null-terminator (e.g. when the string is passed to unsafe system functions). We're using this feature in our migration towards `string_view`-only code, as we can just try swapping the `zstring_view` with a `string_view` to see if there are zero usages of `c_str()` (which is missing from `string_view`).

Moreover, `zstring_view` can be used to return an internal string instead of returning a `const string&` (typing too strict), or returning a `string_view` (no guarantee for null terminator).

Returning `zstring_view` gives more flexibility about internal representation of null-terminated strings: it will still behave very similarly to a `const std::string&`, without risking accidental implicit copies.

The final goal of our project is to convert all the API to receive `string_view` as parameter, and make them return instances of `zstring_view` when they were returning `string`.

`zstring_view`, however, proved to be useful when initially migrating an API without changing the implementation too much: just replace the `const char*` parameter with a `zstring_view`, change the internal code to use the parameter with `<variable>.c_str()`, and then change the usages.
Example

```
void f(const char*x) {
  external_library::g(x);
}

void f1() {
  std::string a1 = ...;
  f(a1.c_str());
}

void f2() {
  constexpr char a2[] = "test";
  f(a2);
}

void f3() {
  const char* a3 = <some char*>;
  f(a3);
}
```

When updating the API we will change f to:

```
void f(zstring_view x) {
  external_library::g(x.c_str());
}
```

Consequently the functions `f1` and `f3` need to be changed (`f2` doesn't need any change because a `zstring_view` can be implicitly constructed from a `char[N]`)

```
void f1() {
  std::string a1 = ...;
  f(a1);
```

```
}

void f3() {
  const char* a3 = <some char*>;
  f(zstring_view(unsafe_length, a3));
}
```

In this way, the usage of an unbounded string (`a3`) will remain tracked, while we simplified the invocation for strings (in `f1`).

Once the `f` function is updated (for example using a function `new_g` from our `external_library`), the parameter type can be changed to `string_view`:

```
void f(string_view x) {
  external_library::new_g(x.data(), x.length());
}
```

Conclusion

From our experience, `zstring_view` is an excellent tool for improving string safety in C++ code, and provides a good migration path from old C-style string processing to more modern string representations, overcoming the limitations of the current use of `string` and `string_view`.

References

- P3081R0: Herb Sutter - "Core safety Profiles: Specification, adoptability, and impact"
- P3436R1: Herb Sutter - "Strategy for removing safety-related UB by default"
- P3566: Marco Foco - "You shall not pass char* - Safety concerns working with unbounded null-terminated strings"