P3711R0
Safer StringViewLike Functions for Replacing `char*` strings

Date: 2025-05-15
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: SG23, LEWG
Author: Marco Foco
Contributors: Alexey Shevlyakov, Joshua Kriegshauser
Reply to: marco.foco@gmail.com

Introduction

This document introduces a set of string utility functions that we used in NVIDIA Omniverse Foundation Library, and were key components to remove `char*` (or more generally `CharT*`) strings usage from our codebase replacing them with the implementation proposed in P3566. All the usages that will still need to use the old-fashioned `char*`s will be marked accordingly to P3566, using the proposed `unsafe_length` tag.

Concepts

We rely on the concepts of [`Safe`|`Unsafe`]`StringViewLike`, as defined in P3566R1. `SafeStringViewLike` represents *bounded* `string_view`-like objects (i.e. implicitly convertible to `string_view` as described in P3566), while `UnsafeStringViewLike` represents *unbounded* `string_view`-like objects (implicitly convertible to `string_view` in C++26, but not implicitly convertible in P3566, e.g. `char*`s).

Functions

We define a function as *safe* if it can perform an operation in a bounded fashion, where bounds are defined by one or all of the operands. For example, testing for a prefix (`starts_with`) is a bounded operation if any of the two operands is bounded (the shortest defines the length), while testing for a suffix (`ends_with`) is a bounded operation if and only if the first operand is bounded, while the second can be unbounded (i.e. it can also be a `CharT*` or an unbounded `CharT[]`).
In accordance with P3566, the *unsafe* operations are tagged with the same `unsafe_length` tag introduced in P3566.

Free function `starts_with`

The function is equivalent to `string_view::starts_with`, but can be applied to operands that are `StringViewLike`, but aren't strictly `string_views`.

If the first operand is bounded, the second is either bounded or unbounded:

```
template<UnsafeStringViewLike TString, SafeStringViewLike TPrefix>
bool starts_with(TString&& s, TPrefix&& p) {
  return string_view{forward<TString>(s)}
    .starts_with(forward<TPrefix>(p));
}
```

If the first operand is unbounded, but the second is bounded, the operation is still considered *safe*:

```
template<UnsafeStringViewLike TString, SafeStringViewLike TPrefix>
bool starts_with(TString&& s, TPrefix&& p) {
{
  string_view p1{p};
  // an empty strings can only start with an empty prefix
  if (is_null(s))
    return p1.empty();
  // no terminator between (0..p.size()-1)
    return !string_view::traits_type::find(str, p1.size(), char{}) &&
      string_view::traits_type::compare(str, p1.data(), p1.size()) ==
0;

}
```

Both operands are unbounded

```
template <UnsafeStringViewLike TString, UnsafeStringViewLike TPrefix>
bool starts_with(carb::cpp::unsafe_length_t, TString&& s, TPrefix&&
p)
{
  return string_view{unsafe_length, forward<TS>(s)}
    .starts_with(unsafe_length, forward<TP>(p));
}
```

Free function `ends_with`

The function is equivalent to `string_view::ends_with`, but can be applied to operands that aren't strictly `string_views`. Whenever a `CharT*` value pointing to `nullptr` is passed, we assume an empty string (according to the idea that `basic_string_view<CharT>{(CharT*)nullptr} ==` <an empty string of CharT> as proposed in P3566).

If the first operand is bounded, the operation is safe:

```
template <SafeStringViewLike TString, StringViewLike TSuffix>
bool ends_with(TString&& str, TSuffix&& suffix)
```

If the first operand is unbounded, the operation is unsafe:

```
template <UnsafeStringViewLike TString, StringViewLike TSuffix>
bool ends_with(unsafe_length_t, TString str, TSuffix suffix)
```

Free function `join`

Concatenate a set of strings together.

The return type can be specified, or left unspecified (default is void). If the return type is unspecified, it's assumed to be a specialization of `basic_string<CharT, Traits>`, where the `CharT` is deduced from the arguments, and the `Traits` type is either deduced, or assumed to be `std::char_traits<CharT>`, if it cannot be deduced.

The *safeness* of the operation is defined by the operands. If they're all bounded (e.g. all `SafeStringViewLike`), the join operation is considered safe.

```
template<typename RetType = void, typename... Args>
// Args... are ALL SafeStringViewLike
auto join(const Args... args);
```

If one of the operands is not safe, an `unsafe_length` tag is required:

```
template<typename RetType = void, typename... Args>
// At least one among Args... is NOT SafeStringViewLike
// (but they're all StringViewLike)
auto join(unsafe_length_t, const Args... args);
```

In our implementation we also proposed other functions, such as a concatenation with a separator, and a concatenation for iterators. These functions are not proposed in this document (we suggest a poll for interest).

Free function `is_null_or_empty`

This function is really simple, it just checks if the parameter is null or is a valid pointer pointing to an empty string. This function is useful for `CharT*`, and is safe by accessing just the first element of the string, and only if the string is not nullptr.

```
template<Char T>
bool is_null_or_empty(const CharT* s) {
  return (!s) || (!*s);
```

Conclusion

In this paper we proposed the free function equivalent of a subset of member functions on `string_view`, to operate on `StringViewLike` objects, separating them into their *safe* and *unsafe* counterparts.