P3711R1
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.

Changes

R1

- Added examples in the form of alternative implementations for starts_with, join and is_empty_or_null
- Added poll results for Sofia SG23

R0

- Document creation

Concepts

We rely on the concepts of [`Safe`|`Unsafe`]`StringViewLike`, as defined in P3566R2. `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 (`SafeStringViewLike)`, the second is either bounded or unbounded (`StringViewLike`):

```
template<SafeStringViewLike TString, StringViewLike 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));
}
```

Current implementations

Currently our experience shows that this problem is solved in a number of ways:
The simplest way to do the same currently is (assuming s and p to be the string and the prefix):

```
string_view(s).starts_with(p)
```

P3566 introduces changes to how char* is proposing to (conditionally) deprecate char* constructors from string, string_view and zstring_view, and that will cause this to become:

```
string_view(unsafe_length, s).starts_with(p)
```

This paper builds on top of P3566, and this line would highlight a problem in the implementation (unsafe length computation) even if the code can be implemented without worrying about the unboundedness of `s` if `p` turns out to be bounded.

despite the fact that testing a string for a prefix is bounded if either the string or the prefix are bounded. This per se would be already confusing, but unfortunately we've found a number of ways this has been implemented in other codebases (`len_s` and `len_p` to be the length of `s` and `p` respectively)

Implementation	**Comment**
manual loop	retains boundedness, verbose
`s == strstr(s, p)`	irrelevant lookup in the rest of the string
`strncmp(s, p, len_p) == 0`	requires length computation for `p` if `p` is an unbounded string (`char*`)
`(lens >= len_p) &&` `  (memcmp(s, p, len_p) == 0)`	requires length computation for `p` or for `s` if `p` or `s` is unbounded (`char*`)

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) {
  string_view{str}.starts_with(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) {
  string_view{unsafe_length, str}.starts_with(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 (i.e. all `SafeStringViewLike`), the join operation is considered safe.

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

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

```
template<typename RetType = void, StringViewLike... Args>
  requires /* At least one Args... is NOT SafeStringViewLike */
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).

Current implementations

Current implementations use a very common idiom (cast to string and +), that is as inefficient as common. When concatenating three StringViewLike objects a, b and c, one user would write:

```
result = string(a) + b + c
```

Which results in multiple allocation and copies:

- allocate space for a copy of `a` (`tmp1`)
- copy `a` into `tmp1`
- allocate space for `tmp1+b` (`tmp2`)
- copy `tmp1` into the first part of `tmp2`
- copy `b` into `tmp2`
- allocate space for `tmp2+c` (`tmp3`)
- copy `tmp2` into `tmp3`
- copy `c` into `tmp3`
- assign to `result` (not counted, as it's probably moved or erased)

for a grand total of 3 allocations and 5 copies (two of size len_a, two of size len_a+len_b, and one of size len_c).

Our implementation does exactly one allocation of the right size, and 3 copies.

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);
}
```

Current implementations

We've seen this function implemented in a number of different ways, some of which included `strlen`/`wcslen`, which is an unnecessary unbounded scan of the string.

Previous polls

2025-06-17 Sofia (SG23)

We should promise more committee time to pursuing P3711R0 from a safety and security perspective

SF	F	N	A	SA
2	7	2	2	0

Consensus

Is there interest in: (Note: some people said they did not have sufficient information at this time to express an opinion and voted N)

starts_with/ends_with:

F	N	A
4	7	1

join:

F	N	A
3	7	1

is_empty:

F	N	A
1	3	7

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.

This paper builds on top of P3566R2, but it can also be implemented independently from it. In our opinion, and each of these utility functions can be separated into a specific paper and proposed independently from the others, and from P3566R2.