

P3566R0: You shall not pass `char*` - Safety concerns working with unbounded null-terminated strings

Date: 2025-01-11
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: SG23
Author: Marco Foco
Contributors: Joshua Krieghauser, Alexey Shevlyakov
Reply to: marco.foco@gmail.com

History

R0

Document creation

Abstract

`strings` and `string_views` are often used as a safer alternative to null-terminated strings. Unfortunately they suffer from an implicit assumption at creation/assignment time, and in some of their functions: the presence of a null-terminator in the input sequence. The absence of the null-terminator can currently lead to undefined behavior inside these functions. There are many cases when the length of the sequence can be computed at compile time, and we should save those usages. In some other cases, we can turn potential undefined-behaviors into either well-defined behavior, or "better behaved" undefined behavior (i.e. turning an unbounded string operation into a bounded string operation). In this paper we propose to restrict the usage of constructors and functions taking a `char*` argument in `string` and `string_view`, to improve range-safety in these operations.

Introduction

P3038R0 suggests the use of `string` and `string_view` as substitute for `char*`, and suggests adding range checking to such classes. P3274R0 further clarifies the Ranges profile, banning subscripting of raw pointers, and introducing a checked indexing operator for strings and views.

In an effort to improve safety on our codebase, we independently started implementing the suggestion from P3038R0, and replaced `const char*s` with `string_views` as much as we could in our internal APIs.

We realized that, in order to improve memory safety further, we should limit the implicit construction of `string` and `string_view` from an unsafe `char*`, and only allow construction from types that will bring along the some additional range information (e.g. `char[]`).

Note: For simplicity of notation, we will often mention `string`, `string_view` and `char*`, but the entire discussion is really about `basic_string<CharT>`, `basic_string_view<CharT>`, and `CharT*`.

Proposal

There are a number of other cases in the standard library where null-terminated strings are expected, and, while we aim in the future to address most of them, this proposal will be mainly limited to addressing the issues in `string` and `string_views`, and strictly related usages.

We aim to separate the function that take a naked `char*` in two categories:

- Functions that can be implemented in a safe way (computing them with **bounded** memory access)
- Functions that cannot be implemented safely, and need to deal with **unbounded** memory access (e.g. unbounded scan for determining the string length)

In some cases, we will be able to separate functions of the second category (unsafe) into two functions (one unsafe and one safe):

- The first one, taking a bare `char*` (unsafe) will compute an unbounded string length at run-time
- The second one (safe) will capture the bounded types before they decay (e.g. `char[N]`), and compute the string length only in the safe region (`0..N-1`).

We propose to then `[[deprecated]]` all unsafe usages, and replace them with equivalent versions tagged versions of the same functions (proposed tag: `unsafe_length`, of type `unsafe_length_t`).

Safe functions in `char_traits`

One important aspect of this proposal is the introduction of a new function in `char_traits::length_s`. This function is the bounded counterpart of `char_traits::length_s` and has two overloads

```

template<size_t N>
size_t length_s(const char_type (&s)[N]) {...}

size_t length_s(const char_type* s, size_t N) {...}

```

Both versions behave similarly to `strnlen_s`, returning the number of characters before the null terminator if that appears before the size provided (or implied by the underlying array), or `N` if the terminator was not found.

Changes to `std::string` and `std::string_view`

Constructing and assigning

Construction and assignment from `char*` of both classes requires an unbounded memory scan to determine the string length. At the moment, this constructor is typically used for both bounded strings (`char[]`) and unbounded (`char*`). We want to separate bounded and unbounded cases, keeping the former and deprecating the latter. We will then introduce a tagged replacement for the deprecated functions.

Example for `string_view`

Before:

```

constexpr string_view(const char *p) noexcept : _data(p),
_size(Traits::length(p)) {...}

```

After:

```

[[deprecated]] constexpr string_view(const char *p) noexcept :
_data(p), _size(Traits::length(p)) {...}

```

```

template<size_t N>
string_view(const char (&p)[N]) noexcept : _data(p),
_size(Traits::length_s(p, N)) noexcept {...}

```

```

explicit constexpr string_view(unsafe_length_t, const char *p)
noexcept : _data(p), _size(Traits::length(p)) {...}

```

The bounded-memory-range constructor/assignment will be used when dealing with string literals and strings built within a fixed-size array. In these cases, we will use `N` as the length of the string should no null-terminator be found within the range.

This does not represent a breaking change with respect to status quo, as all the usages with non-null-terminated `char` sequences would currently result in undefined behavior (out of bounds access), and we're just giving a well-defined behavior to this operation.

Member function: `copy`

The `copy` member functions of `string` and `string_view` is **bounded** by the current object's length and the count of characters requested, and is therefore considered safe

Member function: `compare` and operator `<=>`

The only potentially unsafe member function has signature:

```
constexpr int compare(const char* s) const;
```

This member function does not require any unbounded operation because it will exit as soon as the first difference is encountered.

It will compare the first `size()` characters of both sequence, and only if they're all equal, it will check the `size()+1` character (`s[size()]`), to verify the sequence `s` terminates correctly.

The non-member overloads of the operator `<=>` can all be defined in terms of the `compare` member function (exactly as today).

Member function: `starts_with`

The potentially unsafe member function has signature:

```
constexpr bool starts_with(const char* s) const;
```

This member function does not require any unbounded operation because it will exit as soon as the first difference is encountered.

It will compare at most `size()` characters from the sequence `s` (as it doesn't need to verify that the sequence is terminating).

Member function: `ends_with`

The potentially unsafe member function has signature:

```
constexpr bool ends_with(const char* s) const;
```

This member function does not require any unbounded operation because it can compute `sz = length_s(s, size()+1)`

- If the result is `size()+1` the provided suffix is longer than the current object, and the result is false
- If the result is smaller, we can compare the sequences by returning `ends_with(string_view(s, sz))`

It will visit at most `size()+1` characters from the sequence `s`.

Member function `contains`

The potentially unsafe member function has signature:

```
constexpr bool contains(const char* s) const;
```

This member function does not require any unbounded operation because it can compute `sz = length_s(s, size()+1)`

- If the result is `size()+1` the provided suffix is longer than the current object, and the result is false
- If the result is smaller, we can compare the sequences by returning `contains(string_view(s, sz))`

It will visit at most `size()+1` characters from the sequence `s`.

Member function `find` and `rfind`

The potentially unsafe member functions has signatures:

```
constexpr size_type [r]find(const char* s) const;
```

This member function does not require any unbounded operation because it can compute `sz = length_s(s, size()+1)`

- If the result is `size()+1` the provided suffix is longer than the current object, and the result is false
- If the result is smaller, we can compare the sequences by returning `[r]find(string_view(s, sz))`

It will visit at most `size()+1` characters from the sequence `s`.

Member Functions `find_first_of`, `find_last_of`, `find_first_not_of` and `find_last_not_of`

The unsafe member functions has signature (example with `find_first_of`):

```
constexpr size_type find_first_of(const char* s, size_type pos = 0)
const;
```

Here there's no way to deduce an upper bound for the length of `s`. Like in the construction/assignment case, we have to split these usages, deprecate the unsafe versions, and add the tagged member functions:

Example with `find_first_of`

```
[[deprecated]] constexpr size_type find_first_of(const char* s,
size_type pos = 0) const;

template<size_t N>
constexpr size_type find_first_of(const char (&s)[N], size_type pos =
0) const noexcept;

constexpr size_type find_first_of(unsafe_length_t, const char* s,
size_type pos = 0) const;
```

Changes to `std::string` only

Member function `insert`

Unsafe member function:

```
constexpr string& insert(size_type index, const char* s);
```

It's impossible to deduce an upper bound for the length of `s`, so we deprecate the member function with the usual outcome:

```
[[deprecated]] string& insert(size_type index, const char* s);

template<size_t N>
constexpr string& insert(size_type index, const char (&s)[N]);

constexpr string& insert(unsafe_length_t, size_type index, const char*
s);
```

Member function `append` and operator `+=`

Unsafe member functions:

```
constexpr string& append(const char* s);
constexpr string& operator +=(const char* s);
```

It's impossible to deduce an upper bound for the length of `s`, so we deprecate the member function with the usual outcome:

```
[[deprecated]] constexpr string& append(const char* s);  
[[deprecated]] constexpr string& operator +=(const char* s);
```

```
template<size_t N>  
constexpr string& append(const char (&s)[N]);  
template<size_t N>  
constexpr string& operator +=(const char (&s)[N]);
```

```
constexpr string& append(unsafe_length_t, const char* s);
```

No tagged replacement can be offered for the operator `+=`

Member function `replace`

Unsafe overloads:

```
constexpr string& replace(size_type pos, size_type count, const char*  
cstr);  
constexpr string& replace(const_iterator first, const_iterator last,  
const char* cstr);
```

It's impossible to deduce an upper bound for the length of `s`, so we deprecate the member function with the usual outcome:

```
[[deprecated]] constexpr string& replace(size_type pos, size_type  
count, const char* cstr);  
[[deprecated]] constexpr string& replace(const_iterator first,  
const_iterator last, const char* cstr);
```

```
template<size_t N>  
constexpr string& replace(size_type pos, size_type count, const char  
(&s)[N]);  
template<size_t N>  
constexpr string& replace(const_iterator first, const_iterator last,  
const char (&s)[N]);
```

```
constexpr string& replace(unsafe_length_t, size_type pos, size_type  
count, const char* cstr);  
constexpr string& replace(unsafe_length_t, const_iterator first,  
const_iterator last, const char* cstr);
```

Non-member operator+

Unsafe overloads:

```
constexpr string operator+(const string& lhs, const Char* rhs);
constexpr string operator+(const char* lhs, const string& rhs);
constexpr string operator+(string&& lhs, const char* rhs);
constexpr string operator+(const char* lhs, string&& rhs);
```

It's impossible to deduce an upper bound for the length of `s`, so we deprecate the member function with the usual outcome:

```
[[deprecated]] constexpr string operator+(const string& lhs, const
Char* rhs);
[[deprecated]] constexpr string operator+(const char* lhs, const
string& rhs);
[[deprecated]] constexpr string operator+(string&& lhs, const char*
rhs);
[[deprecated]] constexpr string operator+(const char* lhs, string&&
rhs);
```

```
template<size_t N>
constexpr string operator+(const string& lhs, const char (&rhs)[N]);
template<size_t N>
constexpr string operator+(const char (&lhs)[N], const string& rhs);
template<size_t N>
constexpr string operator+(string&& lhs, const char (&rhs)[N]);
template<size_t N>
constexpr string operator+(const char (&lhs)[N], string&& rhs);
```

No tagged replacement can be offered for the `operator +`

Changes to `std::zstring_view`

As a note for `zstring_view` described in P3081, will need to receive all the changes coming from `string_view`.

In addition, `zstring_view` will have to guarantee the presence of the null-terminator when built from bounded-ranges. This can be achieved by computing `length_s` on the provided sequence, and verifying that the effective length returned is less than $N-1$ (with N being the number of characters in the sequence).

Alternatives

- If we want to frame this proposal in the "profiles" framework, we propose to introduce a new annotation `[[ranges_deprecated]]` which will be used when passing unbounded memory ranges. This will deprecated the offending constructor selectively, without incurring in ODR violations (we evaluated other options, where the offending functions were "disappearing" under the ranges profile, but that would generate ODR violations in codebases that mix different profile configurations)
- Another alternative is the direct removal of any function marked as `[[deprecated]]` in this document

Conclusion

In this paper we proposed to restrict the usage of constructors and functions taking a `char*` argument in `string` and `string_view`, with the scope of improving range-safety of these operations.

The changes proposed in this document allow to remove or mitigate the effects of undefined behavior in `string` and `string_view`.

Appendix A

Resources on safe C++

- [Bjarne Stroustrup :: Approaching C++ Safety - YouTube](#)
A presentation at Core C++ 2023 where Stroustrup present the idea of a "profile"
- [P2816R0](#): Bjarne Stroustrup, Gabriel Dos Reis - "Safety Profiles: Type and resource Safe programming in ISO Standard C++"
- [P3274R0](#): Bjarne Stroustrup - "A framework for Profiles development"
- [P3081R0](#): Herb Sutter - "Core safety Profiles: Specification, adoptability, and impact"
- [P3436R1](#): Herb Sutter - "Strategy for removing safety-related UB by default"