P3566R2

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

Date: 2025-06-11
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: SG23
Authors: Marco Foco, Joshua Krieghauser, Alexey Shevlyakov, Giuseppe D'Angelo
Contributors:
Reply to: marco.foco@gmail.com

History

R2

- Added reference to P3655R2
- Added discussion about `copy` overloads
- Added more experiments with QT
- Clarified that treating null pointers as empty strings is a proposed change (moved from introduction to specific chapter)

R1

- Introduction of SafeStringView and UnsafeStringView concepts
- Impact on existing code
- Null pointer is empty string
- More on implementation experience (both internal and from Giuseppe D'Angelo)
- Added reference to string_ref paper

R0

Document creation

Abstract

`string`s and `string_view`s 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, which shouldn't be ignored. 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`, `string_view`, and `zstring_view` to improve range-safety in these operations (together with some other minor improvements).

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_view`s as much as we could in our internal APIs.

As part of our roadmap we also implemented `zstring_view`, in a similar way as described in P3655. In R2 of that paper we collaborated with the original authors to include our experience and consideration in using `zstring_view` for migrating an existing large codebase.

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

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>`, `basic_zstring_view<CharT>`, and `CharT*`.

Also, right we noticed that passing a null pointer to any of the functions results in Undefined Behavior. In order to reduce the UB cases, we propose, in all functions, to assume that null pointers represent an empty string, and act accordingly. In our implementation of

`zstring_view`, an internal empty string is used (to ensure the presence of the null-terminator).

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 make `[[deprecated]]` all unsafe usages, and replace them with equivalent 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` and has two overloads

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

constexpr 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`, `std::string_view` and `std::zstring_view`

`zstring_view` is described in P3081, P3710R0 and P3655R1.
The last two papers have been unified into P3655R2.

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[N]`) and unbounded (`char*, 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)) 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.

In addition, `zstring_view` guarantees 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).

Member function: `copy`

The `char*` parameter passed to the copy member functions of `string`, `string_view` and `zstring_view` is **bounded** by the current object's length and the count of characters requested, and is therefore considered safe.
We also propose to introduce overloads of this function with char[N], see the section "New `copy` Overloads" later in this document.

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

The 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 sequences, 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 have 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 sequence 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;
```

It's impossible to deduce an upper bound for the length of s. As with the construction/assignment case, we must 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*
s);
constexpr string& replace(const_iterator first, const_iterator last,
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& replace(size_type pos, size_type
count, const char* s);
[[deprecated]] constexpr string& replace(const_iterator first,
const_iterator last, const char* s);

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* s);
constexpr string& replace(unsafe_length_t, const_iterator first,
const_iterator last, const char* s);
```

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+`

Null pointer is empty string

Right now, passing a null pointer to any of the functions results in Undefined Behavior.
In order to reduce the UB cases, we propose, in all functions, to assume that null pointers represent an empty string, and act accordingly.

New `copy` overloads (not proposed yet)

To further improve safety, we considered adding overloads taking a `char[N]`, that can implicitly compute the range size.

```
template <size_t N>
```

```
constexpr size_type copy(const char (&dest)[N], size_type pos = 0)
const
{
  copy((char*)dest, pos);
}
```

Unfortunately this member function would be disregarded, because of the decay `char[N] -> char*` that would pick up the pre-existing overload, with an incorrect count argument. Implicitly excluding `char[N]` from the existing overload would change the meaning of existing code that relies on `char[N] -> char*` conversion.

Therefore we consider adding a new function retaining the count parameter which would allow disambiguation between the `char*` version of the function, and allow a precondition that would verify that the count specified is within the bounds of the array:

```
template <size_t N>
constexpr size_type copy(const char (&dest)[N], size_type count,
size_type pos = 0) const
  pre(count <= N)
{
  copy((char*)dest, count, pos);
}
```

In order to keep the proposal as similar as possible to the previous version, we're not proposing this in this version, but we explicitly ask for feedback (in the form of a poll).

Concepts

In order to better represent the types expressed here, we introduce some new informal concepts:

- `StringViewLike`
  - Remains unchanged from C++26, i.e. StringViewLike is any type implicitly convertible to string_view AS DESCRIBED IN THE C++26 STANDARD.
- `SafeStringViewLike`
  - StringViewLike is any type implicitly convertible to string_view AS DESCRIBED IN THIS DOCUMENT.
  - SafeStringViewLike<T> is true for all types T that are implicitly convertible to string_view as described in this paper: doesn't accept CharT*
- `UnsafeStringViewLike`
  - StringViewLike && !SafeStringViewLike
  - UnsafeStringViewLike<T> is true for all types T that are implicitly convertible to old string_view, but are not SafeStringViewLike

Implementation Experience

NVIDIA/Omniverse Experience

We implemented the changes proposed in this paper in our Foundation library of the Omniverse project in NVIDIA, together with other changes proposed in P3710R0/P3655R2 (`zstring_view`) and P3711R0 (string utility functions).
These tools proved to be useful to migrate an old codebase to a new, safer string standard, by providing the following:

- Reducing the use of unsafe C string functions (`strlen`, `strchr`, `strstr`, `strcpy`, ...)
- Creating a clear migration to a safer standard for our code and our APIs:
    - API input parameters
        - Long term goal: switch all the input parameters to be string_view (ideally)
        - Short term: migrate the input parameters to be either string_view (if possible) or zstring_view (always possible, see P3710).
    - API returns
        - replace `const string&` return types to `zstring_view`.
        - Only use `string_view` as return type for returns that do not guarantee null termination (e.g. substrings)
    - Code
        - Replace implicit conversions need to be replaced with explicit `unsafe_length`-tagged conversions
        - Replace internal use of char* to string_view (if possible) or zstring_view
        - Change the way string literals are declared. Instead of using `const char* x = "...";` (a C++26 StringViewLike, but unfortunately UnsafeStringViewLike), we use `[static] constexpr char x[] = "...";`, as the latter will retain the bounds in the type (making it a SafeStringViewLike, and usable in string_view described in this paper, but also zstring_view described in P3710 and string utility function described in P3711).

Despite starting in 2017, our project started with a very C-oriented mindset (original target was C++11), migrated to C++14 in 2019, and only recently ported to C++17 (late 2024). This resulted in a lot of code still using plain old C strings. Nonetheless, only few of the usages required an explicit unsafe_length-tagged cast, as many of our use-cases used string literals or constant strings, that were treated separately with the aforementioned search & replace of `const char* x = "...";` to `constexpr char x[] = "...";`, which can be applied automatically to an entire codebase.

Giuseppe D'Angelo Qt Experience

Giuseppe D'Angelo tested the changes on the Qt Library (excluding tests and examples).

Qt has many string and string view classes: `QString` and `QStringView` (UTF-16), `QByteArray` and `QByteArrayView` (raw bytes, à la std::string), `QLatin1StringView` (mostly used as a wrapper for string literals), and recently even `QUtf8StringView` (UTF-8) and `QAnyStringView` (type-erased view over Latin1, UTF-8, UTF-16). These classes have very complicated overload sets for their constructors, mostly for historical and compatibility reasons.

Qt string(view) classes already accept null pointers, and create empty string(view)s.

Applying the changes of this paper to the various string(view) classes yields mixed results:

- `QString` has a constructor from `const char *`, but it's already disabled during Qt's own build. This is something that also users can do for their own projects. The reason for this ability is to prevent encoding mistakes: the constructor expects data encoded as UTF-8, but it's easy to accidentally pass data in other encodings. To aid users `QString` already also has a constructor that takes a `const char[N]` (like what is proposed in this paper), to deal with string literals; and a `QString::fromUtf8(QByteArrayView)` factory function that clearly indicates what is the expected encoding of the data.
- Introducing the proposed deprecations for `QStringView` deprecations resulted in very few breakages, around 200LOC, most of which were fairly mechanical to fix.
- The real pain point is introducing the deprecations into `QByteArray` and `QByteArrayView` as those classes are widely used when dealing with `char` data. Several hundred cases were hit by the just-introduced deprecation notices. Some notes:
    - Many times the length of the data was actually available (e.g. data coming from compile-time tables), but it was just "lost in the way". Of course one could switch to the new constructor, but a better solution would be to refactor the functions that read data out these tables so that they yield views (i.e. carry the size) rather than raw pointers.
    - A significant percentage of cases were just API mistakes, e.g. a `QByteArray` was passed to a function taking a `QByteArrayView` but data() was called on the array. It's likely that the function signature was changed from taking `const char *` to take a `QByteArrayView`, but the call site was never changed.
    - The entire translation system is based on macros and const char *, although most of the time these are just string literals in the source code. Changing these APIs to use e.g. `QByteArrayView` may introduce too many source incompatibilities.
    - Similarly, for historical reasons, all of Qt's runtime reflection APIs yield const char *, causing many warnings that one is unable to quickly address. One could concoct adding new APIs and migrating usages, but that's certainly a non-trivial amount of work; it's unclear how much of it could be automated by e.g. a Clazy check.

This experiment also led to an interesting observation: certain language constructs force decaying from string literals (arrays) to pointers; notably, the ternary operator and initializer lists. A snippet like this:

```
void f(std::string_view);
f( cond ? "first" : "second" );
```

is going to decay the arguments into pointers and then call the (now) deprecated constructor of std::string_view. There is a straightforward workaround, which is deploying user-defined literals:

```
f( cond ? "first"sv : "second"sv );
```

What Giuseppe has observed in Qt is that this actually results in a better code generation: calls to `strlen` that were present in the original version disappear once the UDLs are used (example on Compiler Explorer).

Alternatives

In P3566R0 we discussed the alternative of applying the changes under profile. This wasn't polled, and was left as an exploration. Now that profiles will be delivered as whitepaper and not as a core feature, we want to reevaluate the matter, and understand the direction the committee prefers, polling the two alternatives (see "Proposed polls" section).

Proposed Polls

We propose a few polls for continuing this effort in a direction that is aligned with the committee:

- Whether the committee wants to see this proposal connected to the profiles, having the char* constructor to be *conditionally* deprecated under some security profile, or if we want to deprecate those constructors in C++29, and suggest the users to fix their code by using explicit unsafe casts (replacing cases where implicit `char* -> string[_view]` cast is happening, e.g. changing calls like `f(x)`, to explicit casts to `f(string_view(unsafe_length, x))`.
    - If we want to frame this proposal in the "profiles" framework, we propose to introduce a new annotation (e.g. `[[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
- About the "treat null pointers as empty string" option
    - Keep in this proposal, removing the precondition on all functions
    - Only apply if preconditions are disabled
    - Move to a different paper, expand the scope to the entire
- Include the "New copy overloads" section in the proposal?

Conclusion

In this paper we proposed to restrict the usage of constructors and functions taking a `char*` argument in `string`, `string_view`, and `zstring_view` with the scope of improving range-safety of these operations.
In addition to it we proposed to treat all null-pointers in these classes as if they were empty strings.
Changes of the same nature were also proposed for methods in the same classes.

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

Appendix A

Resources on safe C++

- Bjarne Stroustrup :: Approaching C++ Safety - YouTube
  A presentation at Core C++ 2023 where Stoustrup 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"
- N3442: Jeffrey Yasskin - "String_ref: a non-owning reference to a string"
- P3655R2: Peter Bindels, Hana Dusíková, Jeremy Rifkin, Marco Foco, Alexey Shevlyakov - "std::zstring_view"