Proposal for C2y

WG14 N3289

Title: Standardize strnlen and wcsnlen (v2)

Author, affiliation: Christopher Bazley, Arm. (WG14 member in individual capacity – GPU expert.)

Date: 2024-06-21

Proposal category: Feature

Target audience: General Developers, Library Developers

Abstract: This paper proposes that the POSIX functions strnlen and wcsnlen be incorporated into the C standard.

Prior art: POSIX, GNU C library, Linux.

Standardize strnlen and wcsnlen (v2)

Reply-to: Christopher Bazley (chris.bazley@arm.com)
Document No: N3289
Date: 2024-06-21

Summary of Changes

N3252

- Initial proposal

N3289

- Added an alternative example implementation of `strnlen` that uses `memchr`.
- Revised the proposed wording changes for `strnlen` and `wcsnlen` and the associated rationale.
- Added possible wording changes for `wcsncat`, `strnlen_s` and `wcsnlen_s` to align their descriptions with the new wording proposed for `strnlen` and `wcsnlen`.

Rationale

The requirement to determine the bounded length of a string is a common one. Consequently, many C libraries provide the following functions to calculate that result for ordinary and wide character arrays:

```
#include <string.h>
size_t strnlen(const char s[.maxlen], size_t maxlen);
```

(strnlen(3) — Linux manual page [1])

```
#include <wchar.h>
size_t wcsnlen(const wchar_t s[.maxlen], size_t maxlen);
```

(wcsnlen(3) — Linux manual page [2])

These functions have been part of the GNU project's C library since at least 2001 [3] and later became part of the POSIX.1-2008 standard.

Instead of standardizing the `strnlen` and `wcsnlen` functions, WG14 chose to standardize similarly named functions as part of Annex K of the C11 standard:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
size_t strnlen_s(const char *s, size_t maxsize);
```

(K.3.7.4.4 The `strnlen_s` function, ISO/IEC 9899:2023)

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
size_t wcsnlen_s(const wchar_t *s, size_t maxsize);
```

(K.3.9.2.4.1 The `wcsnlen_s` function, ISO/IEC 9899:2023)

Like the rest of Annex K, `strnlen_s` and `wcsnlen_s` are an optional extension. Implementers are under no obligation to provide them, and most (including Clang and GCC) do not. In turn, this is likely to have hindered adoption, because programmers cannot rely on those functions being present.

The rationale given for adding these functions was:

> *The strnlen_s function is useful when dealing with strings that might lack their terminating null character. That the function returns the number of elements in the array when no terminating null character is found causes many calculations to be more straightforward. The technical report itself uses strnlen_s extensively in expressing the runtime-constraints of functions.*
>
> *The strnlen_s function is identical [sic] the Linux function strnlen*

(Rationale for TR 24731 — Extensions to the C Library — Part I: Bounds-checking interfaces [4])

Why standardize a function that "is identical" to an already established function with a different name? (As a matter of fact, `strnlen_s` and `strnlen` are not quite identical: the latter has undefined behaviour if passed a null pointer.)

In N1967, Sebor and O'Donell proposed that Annex K be either removed from the next revision of the C standard or deprecated and then removed [5]. Periodically, the question of whether to follow through on this idea arises in the committee.

However, `strnlen_s` and `wcsnlen_s` differ from other Annex K functions:

> *Except for strnlen_s and wcsnlen_s, functions in the technical report have a “Runtime-constraints” section that lists a series of “shall” or “shall not” statements that the program must satisfy when calling a library function.*

(Rationale for TR 24731 — Extensions to the C Library — Part I: Bounds-checking interfaces)

This means that implementors could easily provide `strnlen_s` and `wcsnlen_s` without concerning themselves with the thread-safety issues of runtime constraint handling which were described by Seacord in N2809 [6].

`strnlen_s` and `wcsnlen_s` also differ from most other Annex K functions in that they do not fully or partially duplicate the functionality of other standard functions. It would therefore be beneficial to users for their functionality to be mandatory, not an optional extension.

According to Annex K itself:

> *This annex provides alternative library functions that promote safer, more secure programming.*

(K.1 Background, ISO/IEC 9899:2023)

In the opinion of this author, the inclusion of `strnlen_s` and `wcsnlen_s` in Annex K was a category mistake. Those functions do not exist primarily to serve the needs of secure programming: they are foundational to solving many problems.

For example, it is impossible to implement the `strndup` function efficiently without `strnlen_s`, or code resembling it. The only alternative to iterating over the passed-in array to find the terminating null character would be to always allocate enough storage for the specified maximum number of characters. That will have a bad outcome if a caller passes `SIZE_MAX` (e.g., because `strdup` is implemented in terms of `strndup`).

As noted by the authors of TR 24731, many operations cannot be economically described without recourse to a function resembling `strnlen`. That is also what motivated this paper: It’s absurd to describe behaviour in terms of `strnlen_s` and `wcsnlen_s` (because those are the standard functions), only to have to search and replace those names with `strnlen` and `wcsnlen` to allow the code to be translated by a real implementation.

It could be argued that `wcsnlen` is less commonly used than `strnlen` and therefore does not merit inclusion. This author believes such irregularities (including the omission of `wcsdup` from C23) merely serve to provoke surprise and irritation from users, without significantly reducing the burden on implementers. It is also hard to reconcile omission of `wcsnlen` with continued inclusion of `wcsnlen_s` in the C standard, since space is thereby used describing a function that usually does not exist, instead of one which usually does.

Implementation

The strnlen and wcsnlen functions can be implemented using only a few lines of code but are not trivial to implement correctly. It is not beneficial to force strictly conforming programs to reinvent this wheel.

For example, the following implementation is subtly broken because s[p] is evaluated before p < n:

```
#include <stddef.h>

size_t strnlen(const char *s, size_t n)
{
    size_t p = 0;
    while (s[p] && p < n)
        p++;
    return p;
}
```

It sometimes erroneously reads n+1 characters instead of no more than n characters. The effects of this error would not be observable at runtime except when operating on an array that contains no null character, and probably not even then.

The following implementation [7] is believed to be correct:

```
#include <stddef.h>

size_t strnlen(const char *s, size_t n)
{
    size_t p;
    for (p = 0; p < n && s[p]; p++) {}
    return p;
}
```

The correct implementation is reasonably efficient for most modern CPU architectures. For example, the following translation is generated by Clang 18.1.0 for an Arm Cortex-M4 embedded processor:

```
strnlen:
        cmp     r1, #0
        itt     eq
        moveq   r0, #0
        bxeq    lr
        mov     r2, r0
        movs    r0, #0
.LBB0_2:                                @ =>This Inner Loop Header: Depth=1
        ldrb    r3, [r2, r0]
        cmp     r3, #0
        it      eq
        bxeq    lr
.LBB0_3:                                @   in Loop: Header=BB0_2 Depth=1
        adds    r0, #1
        cmp     r1, r0
        bne     .LBB0_2
        mov     r0, r1
        bx      lr
```

It is alternatively possible to implement `strnlen` by using `memchr` to search for a null character in the given array [8]. This may be more efficient than a standalone implementation of `strnlen`, depending on how well `memchr` has been optimised:

```
#include <string.h>

size_t strnlen(const char *s, size_t n)
{
    const char *p = memchr(s, '\0', n);
    return p == NULL ? n : p - s;
}
```

The translation generated by Clang 18.1.0 for an Arm Cortex-M4 is also slightly shorter than that generated for the standalone implementation of `strnlen`:

```
strnlen:
        push    {r4, r5, r7, lr}
        mov     r4, r1
        movs    r1, #0
        mov     r2, r4
        mov     r5, r0
        bl      memchr
        cmp     r0, #0
        it      ne
        subne   r4, r0, r5
        mov     r0, r4
        pop     {r4, r5, r7, pc}
```

Proposed wording changes

The wording proposed is a diff from the September 3, 2022 working draft [11]. Green text is new text, while ~~red~~ text is deleted text.

7.26.6.5 The `strnlen` function

Synopsis

1

```
#include <string.h>
size_t strnlen(const char *s, size_t n);
```

Description

2 The `strnlen` function counts not more than `n` characters (a null character and characters that follow it are not counted) in the array to which `s` points. At most the first `n` characters of `s` shall be accessed by `strnlen`.

Returns

3 The `strnlen` function returns the number of characters that precede the terminating null character. If there is no null character in the first `n` characters of `s` then `strnlen` returns `n`.

7.31.4.3.2 The wcsncat function

Synopsis

1

```
#include <wchar.h>
wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Description

2 The `wcsncat` function appends not more than `n` wide characters (a null wide character and ~~those~~ wide characters that follow it are not appended) from the array pointed to by `s2` to the end of the wide string pointed to by `s1`. The initial wide character of `s2` overwrites the null wide character at the end of `s1`. A terminating null wide character is always appended to the result.[423)]

Returns

3 The `wcsncat` function returns the value of `s1`.

7.31.4.7.3 The `wcsnlen` function

Synopsis

1

```
#include <wchar.h>
size_t wcsnlen(const wchar_t *s, size_t n);
```

Description

2 The `wcsnlen` function counts not more than `n` wide characters (a null wide character and wide characters that follow it are not counted) in the array to which `s` points. At most the first `n` wide characters of `s` shall be accessed by `wcsnlen`.

Returns

3 The `wcsnlen` function returns the number of wide characters that precede the terminating null wide character. If there is no null wide character in the first `n` wide characters of `s` then `wcsnlen` returns `n`.

B.25 String handling <string.h>

```
size_t strnlen(const char *s, size_t n);
```

Only if the implementation defines `__STDC_LIB_EXT1__` and additionally the user code defines `__STDC_WANT_LIB_EXT1__` before any inclusion of `<wchar.h>`:

```
size_t strnlen_s(const char *s, size_t ~~maxsize~~ n);
```

B.30 Extended multibyte/wide character utilities <wchar.h>

```
size_t wcsnlen(const wchar_t *s, size_t n);
```

Only if the implementation defines `__STDC_LIB_EXT1__` and additionally the user code defines `__STDC_WANT_LIB_EXT1__` before any inclusion of `<wchar.h>`:

```
size_t wcsnlen_s(const wchar_t *s, size_t ~~maxsize~~ n);
```

K.3.7.4.4 The strnlen_s function

Synopsis

1

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
size_t strnlen_s(const char *s, size_t ~~maxsize~~ n);
```

Description

2 The `strnlen_s` function ~~computes the length of the string pointed to by `s`.~~ counts not more than `n` characters (a null character and characters that follow it are not counted) in the array to which `s` points. At most the first `n` characters of `s` shall be accessed by `strnlen_s`.

Returns

3 If `s` is a null pointer,[515] then the `strnlen_s` function returns zero.

4 Otherwise, the `strnlen_s` function returns the number of characters that precede the terminating null character. If there is no null character in the first `n`~~`maxsize`~~ characters of `s` then `strnlen_s` returns `n`.~~`maxsize`. At most the first `maxsize` characters of `s` shall be accessed by `strnlen_s`.~~

K.3.9.2.4.1 The wcsnlen_s function

Synopsis

1

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
size_t wcsnlen_s(const wchar_t *s, size_t ~~maxsize~~ n);
```

Description

2 The `wcsnlen_s` function ~~computes the length of the wide string pointed to by `s`.~~ counts not more than `n` wide characters (a null wide character and wide characters that follow it are not counted) in the array to which `s` points. At most the first `n` wide characters of `s` shall be accessed by `wcsnlen_s`.

Returns

3 If `s` is a null pointer,[539] then the `wcsnlen_s` function returns zero.

4 Otherwise, the `wcsnlen_s` function returns the number of wide characters that precede the terminating null wide character. If there is no null wide character in the first `n`~~`maxsize`~~ wide characters of `s` then `wcsnlen_s` returns `n`.~~`maxsize`. At most the first `maxsize` wide characters of `s` shall be accessed by `wcsnlen_s`.~~

Rationale for wording

The current wording for the `strnlen_s` and `wcsnlen_s` functions is consistent with that for `strlen` and `wcslen`. In one respect it is *too* consistent: the descriptions of the bounded and unbounded versions of each function are identical:

> *The `strlen` function computes the length of the string pointed to by `s`.*

(7.26.6.4 The `strlen` function, ISO/IEC 9899:2023)

versus

> *The `strnlen_s` function computes the length of the string pointed to by `s`.*

(K.3.7.4.4 The `strnlen_s` function, ISO/IEC 9899:2023)

The initial version of this paper therefore inserted the word 'bounded' into the description of `strnlen` and `wcsnlen` (said to compute the length of a 'bounded string') to disambiguate them from `strlen` and `wcslen`. However, that attempted remediation was insufficient.

The standard does not define the term 'bounded string', but defines 'string' as follows:

> *A string is a contiguous sequence of characters terminated by and including the first null character.*

(7.1.1 Definitions of terms, ISO/IEC 9899:2023)

Reading the descriptions of `strnlen_s` and `wcsnlen_s` with the above definition in mind, it appears that the term 'string' has been misused. In particular:

> *At most the first `maxsize` characters of `s` shall be accessed by `strnlen_s`.*

(K.3.7.4.4 The `strnlen_s` function, ISO/IEC 9899:2023)

Given that the number of characters accessed is limited by the value of `maxsize`, it follows that the input sequence of characters need not be terminated by a null character[1]. In other words, the `s` parameter of `strnlen_s` need not point to a string.

Instead, the `s` parameter of `strnlen_s` should be described as a pointer to an array, like the `s2` parameter of `strncat`:

> *The `strncat` function appends not more than `n` characters (a null character and characters that follow it are not appended) from the array pointed to by `s2` to the end of the string pointed to by `s1`.*

(7.26.3.2 The `strncat` function, ISO/IEC 9899:2023)

[1] Conversely, the fact that a null character might appear within the first `maxsize` characters of the input sequence means that the number of elements in the array pointed to by `s` can be fewer than `maxsize`. It would therefore be incorrect to declare `s` as a variably modified array parameter, `s[maxsize]`.

The POSIX descriptions of `strnlen` [9] and `wcsnlen` do not fall into the same trap of describing the input sequence of characters as a string, but do imply that the array might have a terminating character:

> *The `strnlen()` function shall compute the smaller of the number of bytes in the array to which `s` points, not including any terminating NUL character, or the value of the `maxlen` argument. The `strnlen()` function shall never examine more than `maxlen` bytes of the array pointed to by `s`.*

This aspect of their descriptions has been the subject of a defect report [10].

For the above reasons, my proposed descriptions of the `strnlen` and `wcsnlen` functions are based more on the descriptions of `strncat` and `wcsncat` than the descriptions of `strnlen_s` and `wcsnlen_s` or the POSIX descriptions of `strnlen` and `wcsnlen`. I judged it more important for the C standard to be internally consistent than for its wording to be identical to POSIX.

The wording proposed for `wcsnlen` diverges slightly from the existing description of `wcsncat` because "a null wide character and those that follow it" seems open to misinterpretation, since "those" is intended to mean "wide characters" but could be misunderstood as "null wide characters".

When compared to `strnlen_s` and `wcsnlen_s`, the guarantee about the maximum number of characters accessed by `strnlen` and `wcsnlen` has been moved to the main description of those functions because it does not appear to relate to their return value.

When compared to `strnlen_s` and `wcsnlen_s`, the following caveat was omitted from the description of the return value of `strnlen` and `wcsnlen`:

> *If s is a null pointer, then the `xxxnlen` function returns zero.*

(K.3.7.4.4 The `strnlen_s` function, ISO/IEC 9899:2023)

None of the implementations of `strnlen` that I examined implement special behaviour for a null pointer; to do so would be inconsistent with the behaviour of functions such as `strlen`.

When compared to `strnlen_s` and `wcsnlen_s`, the `maxsize` parameter of `strnlen` and `wcsnlen` was renamed as `n`. This conforms with the equivalent parameter of the `strncpy` and `strncat` functions and avoids potential confusion between a result of the `sizeof` operator and a character count.

The section numbering assumes that describing `strlen` and `strnlen` consecutively is desirable, and similarly that `wcsnlen` should be described as close as possible to `wcslen`. However, it also assumes that renumbering 7.31.4.7.2 "The `wmemset` function" is undesirable. Alternatively, 7.31.4.7.2 and the proposed 7.31.4.7.3 could be swapped so that the descriptions of `wcslen` and `wcsnlen` are consecutive. This would be more intuitive for readers.

References

[1] strnlen(3) — Linux manual page
https://man7.org/linux/man-pages/man3/strnlen.3.html

[2] wcsnlen(3) — Linux manual page
https://man7.org/linux/man-pages/man3/wcsnlen.3.html

[3] Blaming glibc/string/strnlen.c at master · lattera/glibc
https://github.com/lattera/glibc/blame/master/string/strnlen.c

[4] Rationale for TR 24731 — Extensions to the C Library — Part I: Bounds-checking interfaces
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1173.pdf

[5] Updated Field Experience With Annex K — Bounds Checking Interfaces
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1969.htm

[6] Annex K Repairs
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2809.pdf

[7] Compiler Explorer
https://godbolt.org/z/49qaPY4f5

[8] Compiler Explorer
https://godbolt.org/z/bTKEv8dhq

[9] strlen, strnlen - get length of fixed size string
https://pubs.opengroup.org/onlinepubs/9699919799/functions/strlen.html

[10] Austin Group Defect Tracker
https://www.austingroupbugs.net/view.php?id=1834

[11] N3054 working draft — September 3, 2022 ISO/IEC 9899:2023
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3054.pdf