N3567 - Additional String Comparison Functions to Complement ‘strcmp’ (V2)

Author: Yair Lenga

Contact: Yair.lenga@gmail.com

Document Number: N3567

Date: 2025-06-08

Target: C2y

Status: Proposal

Abstract

This paper proposes the addition of three simple and intuitive string comparison functions to the C standard library: compare(streq), prefix compare (strprefix) and suffix compare (strsuffix). These functions will provide boolean results for the most common use cases of strcmp: testing for exact equality, and for the most common uses of strncmp - prefix match and suffix match.

Motivation: streq

The standard function strcmp is widely used in C for comparing strings. However, its interface and semantics are often unnecessarily verbose and error-prone when used to test equality. Common idioms such as:

```
    if (strcmp(a, b) == 0) { ... }

    if (!strcmp(a, b)) { ... }
```

are hard to read, easy to misuse (e.g. accidentally writing strcmp(a, b)), and impose unnecessary cognitive load.

A scan of popular open-source C codebases shows that the vast majority of strcmp calls are used to test for equality, with occasional use for inequality, and only a small fraction for ordering comparisons (<, >, etc.).

In addition, many codebases attempt to optimize multi-value check ‘S on of (“foo”, “bar”, “baz”)’ with constructs like:

```
      if (!(strcmp(s, "foo") && strcmp(s, "bar") && strcmp(a, “baz”))
```

which is much clearer written as:

```
      if (streq(s, "foo") || streq(s, "bar") || streq(s, “baz”))
```

Motivation: strprefix & strsuffix

Scan of popular open-source C codebases for strncmp shows similar findings. The function is mostly used for prefix and suffix match:

```
// hardcoded 7 = strlen(“http://”)
If( strncmp(url, “http://”, 7) )  {}

// Better solution is to avoid hard coding the length
If ( strncmp(url, http_prefix, strlen(http_prefix)) {}

// hardcoded 4 = strlen(“.txt”)
If ( strncmp(filename + strlen(filename) - 4, “.txt”, 4 ) {}

// Buggy: what happens if the suffix is longer than filename ?
If ( strncmp(
      filename + strlen(filename) - strlen(txt_suffix),
      Txt_suffix,
      strlen(txt_suffix)) {}
```

Which will be clearer written as:

```
      If ( strprefix(s, “http://”) )  {}
      If ( strsuffix(filename, “.txt”) ) {}
```

By non-scientific survey the prefix/suffix match represent > ⅔ of use cases for strncmp. In many cases, projects introduce their own wrappers, macros, static inline to avoid the potential errors from hard coding constants.

Proposed Interfaces

In <string.h>, the following two functions are added

```
      bool streq(const char *a, const char *b);
      bool strprefix(const char *s, const char *prefix);
      bool strsuffix(const char *s, const char *suffix).
```

Semantics

For streq

Practically: streq(a, b) matches (strcmp(a, b) == 0)

Expression	Result	Notes
streq(“foo”, “foo”)	True	
streq(“foo”, “bar”)	True	
streq(“foo-bar-baz”, “foo”)	False	
streq(“foo-bar-baz”, “baz”)	False	
streq(“foo”, “”)	False	
streq(“foo”, NULL)	Exception	A, B
streq(NULL, “foo”)	Exception	A, B
streq(NULL, NULL)	Exception	A, B

For strprefix

Expression	Result	
strprefix(“foo”, “foo”)	True	
strprefix(“foo”, “bar”)	False	
strprefix(“foo-bar-baz”, “foo”)	True	
strprefix(“foo-bar-baz”, “baz”)	False	
strprefix(“foo”, “”)	True	
strprefix(“foo”, NULL)	Exception	A, B
strprefix(NULL, “foo”)	Exception	A, B
strprefix(NULL, NULL)	Exception	A, B

For strsuffix

Expression	Result	
strsuffix(“foo”, “foo”)	True	
strsuffix(“foo”, “bar”)	False	
strsuffix(“foo-bar-baz”, “foo”)	False	
strsuffix(“foo-bar-baz”, “baz”)	True	
strsuffix(“foo”, “”)	True	
strsuffix(“foo”, NULL)	Exception	A, B
strsuffix(NULL, “foo”)	Exception	A, B
strsuffix(NULL, NULL)	Exception	A, B

Notes

A. Safe Handling of NULL: An alternative proposal will be to handle NULL values, similar to the way SQL handles them - the answer will always be “False”. However, this approach will be a big detour from “C” philosophy, and will be inconsistent with the behavior of many other “simple” string functions: strcmp, strcpy, … which will (usually) crash.

B. Defined behavior for NULL: Worth noting few str functions are capable of handling NULL. For example, snprintf with NULL can be use to “introspect” expected result length, free(NULL), realloc(NULL) have defined behavior. If/When “C” will add “null-safe” string functions (e.g. by suffix, etc) - it can also apply to those functions.

Naming

Almost all modern programming languages offer built in functions for equality, prefix match and suffix match. Common naming are “equals”, “starts with” and “end with”, adopted to the language naming conventions.:

Language	Equality	Prefix match	Suffix match
Proposal	streq(a, b)	strprefix(s, prefix)	strsuffix(s, suffix)
C++	a == b	s.start_with(prefix)	s.ends_with(suffix)
Java	a.equals(b)	s.startWith(prefix)	s.endWith(suffix)

Python	a == b	s.startWith(prefix)	s.endWith(suffix)
Rust	a == b	s.start_with(prefix)	s.ends_with(suffix)
C#	a == b	s.StartWith(prefix)	s.EndWith(suffix)

Common Libraries	Equality	Prefix Match	Suffix Match
glib	g_str_equals	g_str_has_prefix	g_str_has_suffix
Misc name used	str_eq	strstarts	strends

One of the natural choices can be: “str_equals”, “str_starts_with” and “str_end_with” - very common. This proposal suggest using the shorter names “streq”, “strprefix” and “strsuffix” for few reasons:

1. The proposed names streq, strprefix and strsuffix fall within the reserved namespace for string-related functions (prefix str[a-z]) (POSIX.1), which aligns with standard naming conventions in <string.h>.
2. No other string.h function is currently using snake case names like the stdc_* functions.
3. Some libraries/projects have added similar functions - usually using ‘str_’ name space (or the str[A-Z] namespace) - to avoid potential conflict with potential lib c functions.

Alternative Naming

If the committee believe benefits from using function names similar to other languages (especially C++) - outweigh the potential conflict with existing libraries, the following is suggested.

Proposed Name	Alternative Name
streq	str_equals
strprefix	str_starts_with
strsuffix	str_ends_with

Rationale

- Clarity: streq(a, b) expresses the programmer’s intent more clearly than strcmp(a, b) == 0. Likewise for strprefix and strsuffix.

- Safety: Built-in null-pointer checks reduce the risk of undefined behavior.
- Performance: These functions return boolean directly, simplifying branching.
- Maintainability: Multi-string comparisons and negated logic become simpler.
- Portability: Efficient and portable on all platforms.
- Helper introducing new programmers into the language - avoid common pitfalls.

Implementation

Example portable implementation. Performance can be much better by customizing to specific processor, as done in many implementation - GLIBC, MUSL, …

```
#include <stdbool.h>
#include <string.h>

bool streq(const char *a, const char *b) {
     return strcmp(a, b) == 0;
}

bool strprefix(const char *s, const char *prefix)
{
     size_t prefix_len = strlen(prefix) ;
     Return strncmp(s, prefix, prefix_len) ;
}

Bool strsuffix(const char *s, const char *suffix)
{
     size_t s_len = strlen(s) ;
     size_t suffix_len = strlen(suffix) ;
     If ( s_len < suffix_len ) return false ;
     return strncmp(s+s_len-suffix_len, suffix) ;
}
```

Alternatives Considered

- Macros: Lack type safety, may evaluate arguments multiple times.
- Inline user functions: Duplicated across projects, lack standardization.
- Pointer returning functions - like strstr - I did not find significant use in existing code bases. It will add complexity (to handle char *, vs const char *).

Compatibility Note

The proposed names streq/strprefix/strsuffix fall within the reserved namespace for string-related functions (prefix str[a-z]*), which aligns with standard naming conventions in

<string.h>. This deliberate choice ensures minimal conflict with existing symbols and allows projects that have already defined similar utilities to adopt the standard versions quickly.

Summary

This proposal introduces three intuitive and expressive functions for the most common use cases of string comparison in C. It aligns with modern programming practices while preserving C’s philosophy of minimalism and efficiency.