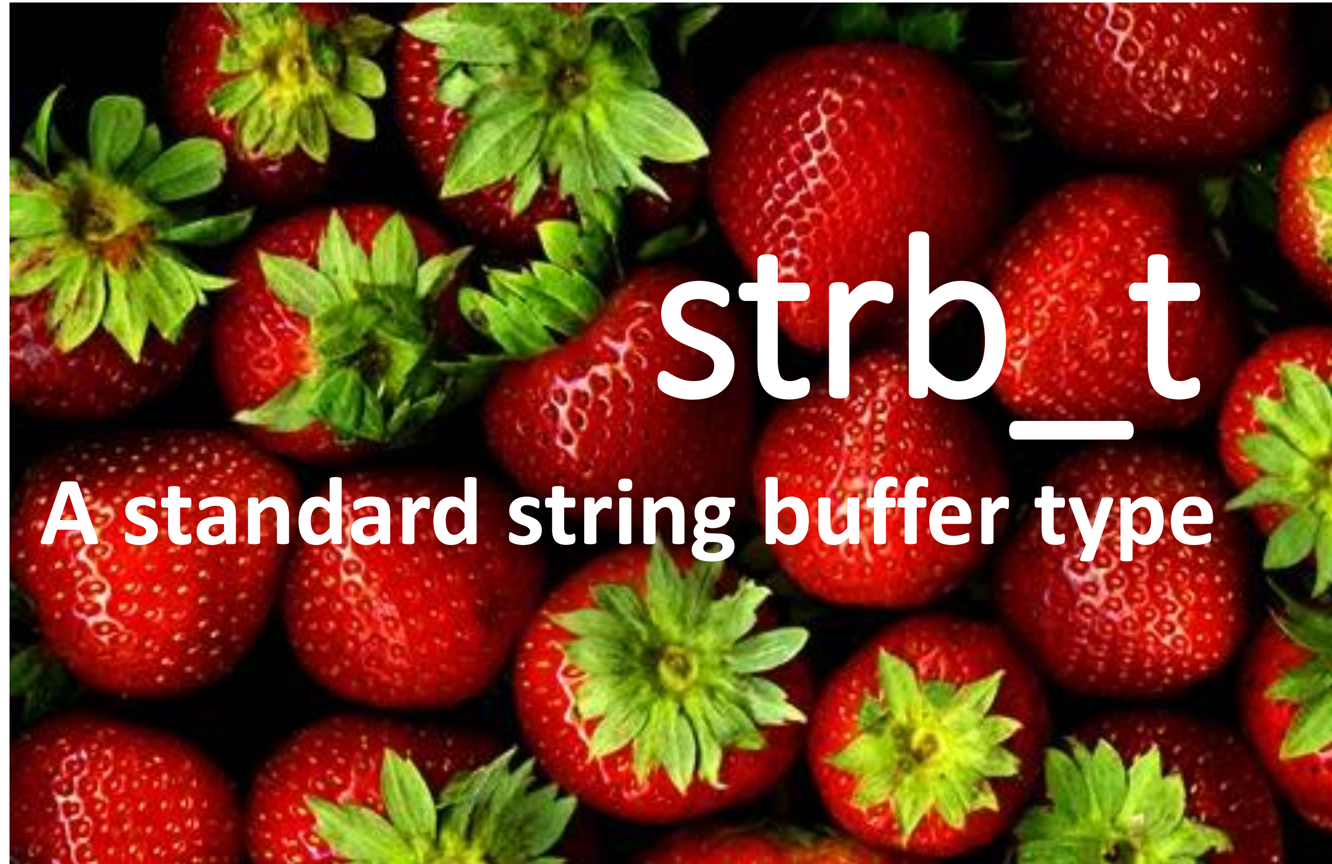


Christopher Bazley
12th June 2024

Rationale

Strings are a fundamental part of every modern programming language and ecosystem.

A lack of standardization in this area has harmed the security and interoperability of code written in C.

This must be addressed to ensure the future viability of the language.

Incorporating safe string functions into the standard would:

- Standardize existing best practice.
- Reduce the level of experience needed to write correct programs.
- Provide a better precedent to follow when users design their own interfaces.
- Make it easier to write interoperable user-designed libraries.

Incorporating safe string functions into the standard would not:

- Need to satisfy all conceivable use-cases optimally.
- Invalidate existing or future user-designed libraries.
- Replace all usage of naked character arrays and pointers.
- Limit choice concerning string allocation and representation.
- Necessitate deprecation or removal of existing standard functions.

Bounds checking is not the solution

- Extra parameters and checks on return values add complexity.
- If the address and size are managed separately, it is easy for callers to accidentally pass the wrong size.
- Users may be tempted to subvert tools used to enforce adherence to secure coding standards by passing a dummy size.
- Encourages use of arrays whose size is determined at compile time but may be insufficient or cause stack overflow.

Truncation can be worse than overflow

"Unintentional truncation results in a loss of data and in some cases leads to software vulnerabilities."

(SEI CERT C Coding Standard)

.

Truncation can be hard to understand

Standard string functions exhibit one of three behaviours:

- No null character is written into the array.
- A null character is written into the last element of the array.
- A null character is written into the first element of the array.

.

Truncation can be hard to understand (2)

Does snprintf…

- guarantee to write a null character at the end of its output?
- require room for a null terminator in its size argument?
- Include a null terminator in its return value?

Even senior engineers get it wrong.

Why not standardize existing functions?

The POSIX functions have:

- An extra object whose lifetime must be managed correctly.
- Surprising behaviour related to unwanted buffering.
- No persistent record of string length. (The file position often reflects this, but it is lost when the stream is closed.)
- No encapsulation of the managed buffer.
- Weak type-safety since a `FILE *` could be a wide-oriented or byte-oriented stream and does not necessarily even manage a string.

Why not standardize existing functions? (2)

The GLib functions have:

- A large, complex interface that relies on in-band special values.
- Highly specialized functions which are bad for composability.
- No encapsulation of the managed buffer.
- No notion of a current position, which complicates consecutive insertions.
- Unsuitable out-of-memory behaviour for many platforms and applications.

Example using standard functions

```
int main(void)
{
    char const *const names[] = {"apple", "orange", "banana", "lime"};
    size_t const data[][6] = {{3,0,2,0,1,0}, {1,2,0,3,3,0}};
    _Optional char *buf = NULL;
    size_t buf_size = 0;
    int err = EXIT_SUCCESS;

    for (size_t i = 0; i < ARRAY_SIZE(data); ++i) {
        for (size_t j = 0; j < ARRAY_SIZE(data[0]); ++j) {
            size_t const len = buf ? strlen(buf) : 0,
                         req = len + strlen(names[data[i][j]]) + 2;
            // +2 for ',' and '\0'

            if (buf == NULL || buf_size < req)
            {
                size_t new_size = buf_size * 2;

                if (new_size < req)
                    new_size = req;

                _Optional char *new_buf = realloc(buf, new_size);
                if (new_buf == NULL) {
                    err = EXIT_FAILURE;
                    break;
                }
                if (buf == NULL) {
                    *new_buf = '\0';
                }

                buf = new_buf;
                buf_size = new_size;
            }

            if (j > 0)
                strcat(buf, ",");

            strcat(buf, names[data[i][j]]);
        }

        if (err) {
            fprintf(stderr, "Failed at %zu (length %zu)\n",
                    i, buf ? strlen(buf) : 0);
            break;
        }

        puts(buf);
        *buf = '\0';
    }

    free(buf);
    return err;
}
```

Example using proposed new functions

```c
int main(void)
{
    const char *const names[] = {"apple", "orange", "banana", "lime"};
    size_t const data[][6] = {{3,0,2,0,1,0}, {1,2,0,3,3,0}};
    _Optional strb_t *sb = strb_alloc(0);
    if (sb == NULL) {
        fprintf(stderr, "Failed at start\n");
        return EXIT_FAILURE;
    }

    for (size_t i = 0; i < ARRAY_SIZE(data); ++i) {
        for (size_t j = 0; j < ARRAY_SIZE(data[0]); ++j) {
            if (j > 0)
                strb_puts(sb, ",");

            strb_puts(sb, names[data[i][j]]);
        }
        if (strb_error(sb)) {
            fprintf(stderr, "Failed at %zu (length %zu)\n",
                    i, strb_len(sb));
            break;
        }

        puts(strb_ptr(sb));
        strb_delto(sb, 0);
    }

    int err = strb_error(sb) ? EXIT_FAILURE : EXIT_SUCCESS;
    strb_free(sb);
    return err;
}
```

Creation/destruction are designed to feel familiar

Create	Destroy	Create	Destroy
strb_alloc	strb_free	calloc	free
strb_dup		strdup	
strb_ndup		strndup	
strb_aprintf		asprintf	
strb_vaprintf		vasprintf	

Position indicator and mode are stateful

Instead of providing variants of every function to insert or overwrite characters at specific positions, both aspects of behaviour are controlled by the `strb_t` object.

```
enum {
  strb_insert,
  strb_overwrite
};

int strb_setmode(strb_t *sb, int mode);
int strb_getmode(const strb_t *sb );

size_t strb_seek(strb_t *sb, size_t pos);
size_t strb_tell(strb_t const *sb );
```

Editing a string resembles writing to a stream

A stateful position indicator and mode (insert/overwrite) are relevant to the following functions:

```
int strb_putc(strb_t *sb, int c);
int strb_nputc(strb_t *sb, int c, size_t n);
int strb_unputc(strb_t *sb);
int strb_puts(strb_t *sb, const char *str );
int strb_nputs(strb_t *sb, const char *str, size_t n);

int strb_vputf(strb_t *sb, const char *format, va_list args );
int strb_putf(strb_t *sb, const char *format, ...);

_Optional char *strb_write(strb_t *sb, size_t n);
void strb_wrote(strb_t *sb);

void strb_delto(strb_t *sb, size_t pos);
```

Why are position and mode stateful?

Because C is not like other languages:

- C programs must be *simple* in order to be *correct*.
- C programs are expected to be *efficient*.
- C programmers should not be expected to manage *many short-lived objects*.

Why are position and mode stateful? (2)

The following function may be inserting, overwriting, appending, or prepending anywhere in a string.
It can be reused by anyone for any purpose, forever.

```
void get_fruit(strb_t *sb, size_t i)
{
    const char *const names[] = {"apple", "orange", "banana", "lime"};
    if (i < ARRAY_SIZE(names))
        strb_putf(sb, "%zu:%s", i, names[i]);
    else
        strb_puts(sb, "unknown fruit");
}
```

Why are position and mode stateful? (3)

```
enum { MONDAY, TUESDAY };
void list_fruit(strb_t *sb, int day)
{
    size_t const days[2][6] = {
        [MONDAY] = {1,2,0,3,3,6},
        [TUESDAY] = {3,0,1,0,2,0}
    };
    if (day >= ARRAY_SIZE(days)) {
        strb_puts(sb, "unknown day");
        return;
    }
    for (size_t j = 0; j < ARRAY_SIZE(days[0]); ++j) {
        if (j > 0)
            strb_puts(sb, ",");

        get_fruit(sb, days[day][j]);
    }
}
```

Why are position and mode stateful? (4)

```
int main(void)
{
    _Optional strb_t *sb = strb_alloc(0);
    if (sb == NULL)
        return EXIT_FAILURE;

    strb_puts(sb, "Monday: ");
    list_fruit(sb, MONDAY);

    strb_puts(sb,"\nTuesday: ");
    list_fruit(sb, TUESDAY);

    puts(strb_ptr(sb));

    int err = strb_error(sb) ? EXIT_FAILURE : EXIT_SUCCESS;
    strb_free(sb);
    return err;
}
```

Convenience functions allow string replacement

The position indicator and mode have no effect on the following functions:

```
int strb_cpy(strb_t *sb, const char *str );
int strb_ncpy(strb_t *sb, const char *str, size_t n);

int strb_vprintf(strb_t *sb, const char *format, va_list args);
int strb_printf(strb_t *sb, const char *format, ...);
```

Error indicator allows deferred error handling

```
bool strb_error(strb_t const *sb );
void strb_clearerr(strb_t *sb);
```

- Writing comprehensive error-handling code is laborious.
- Immediate error-handling makes code harder to read.
- The presence of such code harms efficiency (branch prediction, instruction cache use).
- The performance of code that fails typically doesn’t matter.

Cri de cœur

C is on the verge of being *impossible to defend* as a rational choice in the corporate environment because the software ecosystem is fragmented.

If WG14 chose to, they could begin to fix this for the simple use-cases that all C programmers have.

Cri de cœur (2)

Code written by different people or organizations has to interact at the level of the lowest common denominator, which is error-prone and inefficient.

We can carry on in our walled gardens, but nobody we hire to work on our code will know anything about our interfaces.

New hires may begin by writing junk code that uses `realloc`, `memcpy` and `memmove` directly instead of using appropriate abstractions.

Cri de cœur (3)

Simple questions need simple answers. At the moment, the answer to “how can I concatenate strings in C without buffer overflow?” is “Use realloc”.

That answer sucks.

“Use GLib” is a better answer, but that ties code to do a very simple, universal thing to one part of a highly fragmented software ecosystem.

I’m proposing an alternative future answer: “Use **strawberries**”.

Reaction from colleagues

"Thank you, it's always bothered me how annoying it is to manipulate strings in C!"

"How is this still a thing? Good luck!"

"Is there any way we can follow along and see how this paper progresses and is received? I'm interested. Thanks!"

"The extra level of indirection might rub people the wrong way, but personally I think it's a perfectly reasonable price to pay. You could mention the potential for small-string optimisation? I love that it follows the stream-like pattern. I think that's perfect."

"While I agree with the decision to used a stored error (agree might be a strong word, perhaps the lesser of many evils?) I'm not sure pointing at OpenGL as prior art helps your cause."

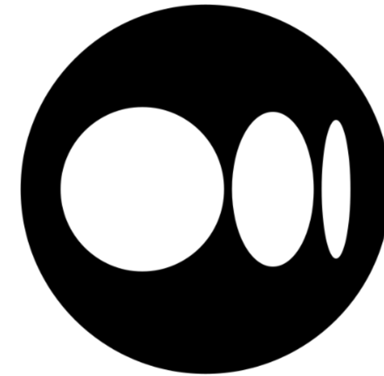
"The API looks nice."

"I'd be honoured to be named in the acknowledgements."

Online reaction

Score of 15
on Reddit.

7 upvotes from
five people.



134 claps from
nine people.