Proposal for C2y

WG14 N3296

Title: strb_t: A standard string buffer type (v2)

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

Date: 2024-07-07

Proposal category: Feature

Target audience: General Developers, Library Developers

Abstract: This paper critiques the standard C library functions, assesses prior art from elsewhere, then proposes a new type and functions to manage strings. Its goal is to eliminate a source of many common programmer errors. The new interface is designed to be as familiar and ergonomic as possible.

Prior art: POSIX, GNU/BSD, Linux kernel, GLib, Arm Mali GPU driver, CBUtilLib.

strb_t: A standard string buffer type (v2)

Reply-to: Christopher Bazley (chris.bazley@arm.com)
Document No: N3296
Date: 2024-07-07

Summary of Changes

N3250

- Initial proposal

N3296

- Introduced a complete type, `strbstate_t` and modified `strb_use` and `strb_reuse` to require a pointer to an object of that type.
- Removed the requirement for freestanding implementations to support dynamic allocation of `strb_t` objects (or the `strb_free` function).
- Renamed `strb_asprintf` and `strb_vasprintf` as `strb_aprintf` and `strb_vaprintf` respectively.
- Removed the `buf` parameter from the declaration of `wcsb_alloc`.
- Modified the meaning of the return value of `strb_seek` and `strb_setmode` to match `fseek` and added a rationale to explain under what circumstances repositioning can fail.
- Removed the requirement for `strb_putf` and `strb_vputf` to return the number of characters generated.
- Functions that were previously specified as returning 'a nonnegative value' or the number of characters written are now specified as returning zero on success.
- Further elucidated the given example of buffer reuse for successive strings.
- Added examples of generating strings with automatic storage duration from a format string.

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.

The language itself provides no operations to deal directly with strings at run time. For example, all three references to "string concatenation" in the index of 'The C Programming Language' (K&R, 1988) concern string literals. This lack of built-in operations is defensible only if standard functions fill in the missing functionality adequately.

Conforming hosted implementations of C must provide most of the standard string functions specified in the headers `<string.h>` and `<stdio.h>`.

Unfortunately, those functions are often misused:

- `strcpy`, `strcat` and `sprintf` can overrun the destination array and provide no mechanism for callers to detect when that is liable to happen.
- `strncpy` does not guarantee termination of its output string by a null character.
- `snprintf` is sometimes recommended as an alternative but it relies on the passed-in size being calculated correctly, and its return value is often ignored or misapplied.

Although C23 adopted `strdup` and `strndup` from POSIX, other functions that would be useful for dealing with strings of unbounded maximum size are absent.

No function is provided to:

- allocate storage for a string generated under control of a format string, as a single operation.
- allocate storage for a copy of one string concatenated with another, as a single operation.
- insert characters into a string, reallocating storage as necessary.

The task of creating safe and powerful string-handling functions has instead been left to users of the language. In contrast, the standard I/O functions provide a relatively high-level abstraction that is fully featured and easy to use correctly (e.g., it's impossible to read or write outside a stream's buffer).

Although static analysis tools can find some buffer overflows caused by misuse of standard string functions, such tools do nothing to repair the reputation or usability 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.

String representation

The C standard defines a string as

> *a contiguous sequence of characters terminated by and including the first null character*

and this definition is implicit in the language's treatment of string literals.

Consequently, operations requiring the length of a string have O(n) complexity.

Another drawback is that there is no efficient standard mechanism for representing substrings ('slices' in some other languages): either a substring must be copied into newly allocated storage to append a null character, or the original string must be modified (as if by `strtok`) to replace one of its characters with null. Similarly, one whole string cannot be prepended to another without its terminator overwriting the first character of the following string.

This paper does not propose any change to that representation, but higher-level abstractions give scope for implementations to differentiate themselves by choosing different trade-offs (for example by storing the length of a string as well as its address). The proposed interface will make it easier for implementers to tailor string handling to specific use-cases or platforms.

Pointer notation

A type qualifier named `_Optional` has been used throughout this paper to clarify declarations and example code. This qualifier was proposed by N3089 [31], which was reviewed by the committee at the Strasbourg meeting in January 2024 with strong consensus to proceed. It can be ignored (for example by defining it as an empty macro) without substantially changing the meaning of the code.

Bounds checking is not the solution

A common approach to guarding against buffer overflows is to require callers to pass an extra parameter specifying the destination array size, when calling a function that writes to an array. Examples include the `snprintf` function, as well as the `strcpy_s` and `strcat_s` functions specified in Annex K of the C standard.

Extra parameters add complexity, as do extra checks on return values. Ironically, such additional complexity may reduce the likelihood of programs being correct. N1969 contains an analysis of flaws in the design and usage of the Annex K functions [29].

If the address and size of a character array are managed separately, it is too easy for callers to accidentally pass the wrong size. This is particularly problematic for strings because of widespread confusion between the size of a string (in bytes, including terminator) and its length (in characters, excluding terminator).

If simplistic tools are used to enforce adherence to secure coding standards by banning 'unsafe' functions in favour of 'safe' alternatives, then users may be tempted to subvert such tools by passing a dummy array size (either the maximum value or an arbitrary value believed to be sufficient):

```
size = strnlen_s(name, (size_t)-1) + 1;
```

The temptation to write code like this is greatest where the true array size is not readily available, hard to compute correctly, or believed to be sufficient because of checks elsewhere.

In the above example, the readability of some code has been damaged by replacing a call to `strlen` but robustness has not improved. Implementing such changes has a significant cost for organizations maintaining large codebases and may be a source of new programmer errors.

In my view, requiring the caller of string-handling functions to pass the available buffer size implicitly encourages use of character arrays whose length is determined at compile time. In some cases, such as when converting a number to a string, this may be appropriate. In many others, such as when concatenating strings of unknown maximum length, it is not.

Modern systems typically handle longer strings than those of the past. In 1983, the maximum filename length in Acorn's Disc Filing System was seven characters, optionally nested within a one-character directory name [1]. By 1985 this had increased to ten characters per file or directory name [2], with arbitrary nesting. By 1999, each individual file or directory name could be up to 255 characters [3].

An existing macro, `FILENAME_MAX`,

> *expands to an integer constant expression that is the size needed for an array of char large enough to hold the longest file name string that the implementation guarantees can be opened or, if the implementation imposes no practical limit on the length of file name strings, the recommended size of an array intended to hold a file name string;*

(7.23.1 of ISO/IEC 9899:2023, Programming languages — C)

When recompiling old software, it is tempting to increase the length of character arrays to some larger (but equally arbitrary) size. Storage is then wasted to allow for the presumed worst-case scenario. If `FILENAME_MAX` has been used to specify array sizes, such changes occur automatically. Given that strings are commonly allocated on the stack and C provides no mechanism to predict or recover from stack overflow, this makes programs fragile.

Passing the size of an array known to be of sufficient size (e.g., because it was dynamically allocated) also incurs runtime costs. Conscientious programmers may feel obliged (or may be required by rules) to add checks on the return value even in such cases:

```
#include <stdio.h>
#include <stdlib.h>
#define PATH_SEP '/'
_Optional char *get_full_path(char const *dir, char const *filename)
{
    int n = snprintf(NULL, 0, "%s%c%s", dir, PATH_SEP, filename);
    if (n < 0) {
        return NULL; // handle error
    }
    _Optional char *path = malloc(n + 1);
    if (path == NULL) {
        return NULL; // handle error
    }
    n = snprintf(path, n + 1, "%s%c%s", dir, PATH_SEP, filename);
    if (n < 0 || (unsigned)n >= n + 1) {
        return NULL; // handle error (impossible?)
    }
    return path;
}
```

Erroneously using `sizeof path` instead of `n + 1` in either of two places in the above example would yield `sizeof(char *)` instead of the intended value. The size of the allocated array would be misrepresented, the buffer overflow check would be broken, or both. This mistake is particularly easy to make if adapting code that previously used an array declaration.

It would be less error-prone to use `sprintf` in place of the second call to `snprintf`. This illustrates why lists of 'banned' functions are a blunt instrument: correctness usually depends on usage and context.

Bounds checks and returned error indications are useless unless the calling code is correct. Inappropriate use of bounds-checked functions can make code worse.

Truncation can be worse than overflow

Many programs seem to be written based on the tacit assumption that string truncation is a safe alternative to buffer overrun. However, truncating a string and continuing to execute a program may cause deferred effects at least as bad as, or worse than, allowing immediate illegal writes.

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

(SEI CERT C Coding Standard [4])

The `strncpy` function does not terminate its output with a null character if truncation occurs. This makes the destination array unsafe for further use as a null-terminated string. Instead, it pads the destination array with null characters:

> *If the array pointed to by s2 is a string that is shorter than n characters, null characters are appended to the copy in the array pointed to by s1, until n characters in all have been written.*

(7.26.2.5 of ISO/IEC 9899:2023, Programming languages — C)

In my experience, this behaviour is rarely what was intended by the programmer. However, `strncpy` does have at least one legitimate use: its behaviour precisely matches the format of a RISC OS sprite [5]. This illustrates again why 'banned' lists are a blunt instrument.

The behaviour of `snprintf` when passed an array of insufficient size differs from `strncpy` in that it ensures the result is null terminated:

> *Otherwise, output characters beyond the n-1st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array.*

(7.23.6.5 of ISO/IEC 9899:2023, Programming languages — C)

`snprintf` is more widely supported than `sprintf_s` and has long been promoted as a 'safe' alternative to `sprintf`. This substitution is typically less harmful than blindly replacing calls to `strcpy` with `strncpy` but can still be problematic.

For example, the following function can unexpectedly delete the wrong file (but only in unusual circumstances which are unlikely to be covered by testing):

```
#include <stdio.h>
#define PATH_SEP '/'
void del_file(char const *dir, char const *filename)
{
    char path[100];
    snprintf(path, sizeof path, "%s%c%s", dir, PATH_SEP, filename);
    remove(path);
}
```

A function such as `snprintf` can be called to generate a truncated string, determine the buffer size required for the full result, or both. Programmers may become confused about whether the returned length is the number of characters generated or written.

Anecdotally, my experience is that even senior engineers are sometimes mistaken about whether `snprintf` guarantees to write a null character at the end of its output (if the passed size is non-zero), and whether its size argument and return value allow room for a null character (one does; the other doesn't).

The following rewrite appears to have solved the truncation issue, but in fact it always truncates the file path to be deleted:

```
#include <stdio.h>
#define PATH_SEP '/'
void del_file(char const *dir, char const *filename)
{
    int n = snprintf(NULL, 0, "%s%c%s", dir, PATH_SEP, filename);
    char path[n];
    snprintf(path, n, "%s%c%s", dir, PATH_SEP, filename);
    remove(path);
}
```

The bounds-checked functions defined in Annex K instead write an empty string to the destination array when a run time constraint violation is detected (including when the array is too small). This can still cause loss of data if not handled carefully (most obviously, the string itself).

In summary, different standard string functions exhibit one of three truncation 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.

We cannot hope to solve the complexities of string handling by education alone. In any case, truncation is rarely correct even when it is supposedly 'safe'.

Effect of bad precedents

It is unfortunate that string handling has been ill-served by the standard library because it plays an important role in establishing norms for user libraries and programs. Consequently, the design of the standard string functions has had a wider negative impact than merely on programs which call those functions.

For example, here is an (anonymized) declaration of real-world function which inserts one or more 64-bit instructions into a buffer:

```
uint32_t foo_insert_bar_op(foo_context *ctx, uint64_t *buf, uint32_t size, uint32_t
operand);
```

Intended usage of this function resembles usage of `snprintf`: it is the responsibility of the caller to update the passed-in values of `buf` and `size` using the return value, which gives the number of array elements consumed.

The free space pointer and record of the amount of remaining space may become separated and may not be updated correctly by every caller. It's easy for callers to accidentally swap the `size` and `operand` parameters. The interface is not type-safe either, given that `*buf` could be any object of type `uint64_t` and `size` could be any integer.

None of these defects are inherent to the language, but most can be observed in its standard library.

Prior art

CBUtilLib

This library [6] has been used by the author since 2012 in many interactive programs that manipulate file paths, menus, and windows. It emphasizes performance and flexibility over encapsulation.

It replaced all ad-hoc string buffer management code but was never intended to replace all usage of character arrays. Its use has helped to eliminate common errors such as buffer overruns and memory leaks.

The amount of implicit state is excessive for many use-cases, but not to the extent that I have felt it necessary to split the structure definition. I tend to use long-lived mutable strings sparingly.

Interface


```
typedef struct
{
  size_t buffer_size, string_len, undo_len;
  _Optional char *buffer;
  char undo_char;
}
StringBuffer;

void stringbuffer_init(StringBuffer *buffer /*out*/);

_Optional char *stringbuffer_prepare_append(
  StringBuffer *buffer /*in,out*/,
  size_t *min_size /*in,out, incl. null*/);

void stringbuffer_finish_append(StringBuffer *buffer /*in,out*/,
                                size_t n /*characters, excl. null*/);

bool stringbuffer_append(StringBuffer *buffer /*in,out*/,
                         const char   *tail /*in*/,
                         size_t        n /*characters, excl. null */);

bool stringbuffer_append_all(StringBuffer *buffer /*in,out*/,
                             const char *tail /*in*/);

bool stringbuffer_append_separated(StringBuffer *buffer /*in,out*/,
                                   char          sep,
                                   const char   *tail /*in*/);

bool stringbuffer_vprintf(StringBuffer *buffer /*in,out*/,
                          const char *format /*in*/,
                          va_list args);

bool stringbuffer_printf(StringBuffer *buffer /*in,out*/,
                         const char *format /*in*/,
                         ...);

void stringbuffer_truncate(StringBuffer *buffer /*in,out*/,
                           size_t len /*characters*/);

size_t stringbuffer_get_length(const StringBuffer *buffer /*in*/);
char *stringbuffer_get_pointer(const StringBuffer *buffer /*in*/);
void stringbuffer_minimize(StringBuffer *buffer /*in,out*/);
void stringbuffer_undo(StringBuffer *buffer /*in,out*/);
void stringbuffer_destroy(StringBuffer *buffer /*in*/);
```

Description

The storage lifetime of the structure is under the client's control, which harms encapsulation but minimizes indirection and allows inline function definitions. The `stringbuffer_init` function cannot fail and is equivalent to default initialisation using `{}`. This could be turned into a useful guarantee.

All write functions (including `stringbuffer_printf`) append to rather than replace the string in the buffer. There is no support for inserting characters and moving the tail of the string, which keeps the interface and implementation simple. No concept of a current insertion point is required. It also avoids the possibility of partially overwriting multibyte characters.

Wide characters (typically UTF-16) can be converted to multibyte characters and appended to the string by using the standard `l` length modifier:

```
wchar_t wstr[] = L"wide string";
stringbuffer_printf(buffer, "%ls", wstr);
```

Additional storage is allocated automatically as the string grows. Functions that allocate storage return a Boolean value to indicate success or failure. This value must be checked immediately (because no error state is stored or exposed).

The user is responsible for calling `stringbuffer_destroy` to free any allocated storage. This is more type-safe than `free` (which cannot be used here) but impedes interoperability with code that needs to take ownership of the underlying character array.

Some functions are strictly redundant:

- `stringbuffer_append_separated(buffer, sep, tail)` is equivalent to `stringbuffer_printf(buffer, "%c%s", sep, tail)`.
- `stringbuffer_append_all(buffer, tail)` is equivalent to `stringbuffer_append(buffer, tail, SIZE_MAX)` or `stringbuffer_printf(buffer, "%s", tail)`.
- `stringbuffer_append(buffer, tail, n)` is equivalent to `stringbuffer_printf(buffer, "%.*s", n, tail)`.

`stringbuffer_get_pointer` returns an unqualified pointer to the first character of the string. This allows interoperability with functions which don't accept `const`-qualified strings, or which temporarily modify a passed-in string. Arguably, this violates encapsulation.

Because the underlying string is accessible, there is no need for many overly specialized functions. For example, concatenation of two `StringBuffer` objects can be implemented as follows:

```
stringbuffer_append_all(dst, stringbuffer_get_pointer(src));
```

Comparison is similarly straightforward:

```
if (strcmp(stringbuffer_get_pointer(a), stringbuffer_get_pointer(b))
{
  // strings mismatch
}
```

So is searching:

```
if (!strchr(stringbuffer_get_pointer(buffer), '_'))
{
  // no underscore
}
```

This saves documentation, implementation, and validation effort, but also means that usage of the interface is more longwinded than it otherwise might be.

`stringbuffer_prepare_append` and `stringbuffer_finish_append` allow direct insertion of strings into the buffer (in cases where the length of a string to be appended is available before the string itself) but they also violate encapsulation. These functions are used to wrap third-party interfaces which cannot be modified to output to a `StringBuffer`.

A crucial detail of `stringbuffer_prepare_append` is that the input value of `*min_size` must include space for a null terminator (like the equivalent parameter of `snprintf`), otherwise insufficient storage may be allocated. The output value of `*min_size` gives the maximum number of characters that can be written, also including any null terminator.

In contrast, the length passed into `stringbuffer_finish_append` does *not* include any null terminator written by the user (like the return value of `snprintf`). A null character is then written after the user-specified length, typically overwriting the last byte of the storage allocated by `stringbuffer_prepare_append`. This allows strings generated by functions that always emit a null terminator to be appended, without requiring one to be present.

Single-step undo is supported by storing the previous length of the string and any character overwritten by the most recent truncation. However, it also relies on the fact that the user controls when to call `stringbuffer_minimize` to free any excess storage. Attempting to undo after calling `stringbuffer_minimize` has no effect.

Without explicit undo:

- Undoing an append (e.g., to generate different file paths based on the same root) could instead be implemented by a combination of `stringbuffer_get_length` and `stringbuffer_truncate` if the previous length were stored externally.
- Undoing truncation (e.g., to create each successive directory of a path) could instead be implemented by violating encapsulation to overwrite the null character at an address relative to that returned by `stringbuffer_get_pointer`.

Usage

Buffer reuse for successive strings

It is often more efficient to initialise a single buffer and reuse it for multiple strings, which also avoids the need for complex error handling code.

```
bool append_to_csv(StringBuffer *const csv, char const *const value)
{
  return (stringbuffer_get_length(csv) == 0 ||
          stringbuffer_append_all(csv, ",")) &&
         stringbuffer_append_all(csv, value);
}

bool build_ships_stringset(StringBuffer *const output_string,
  char const *const graphics_set,
  bool const include_player, bool const include_fighters)
{
  /* Build string suitable to pass to stringset_set_available() */
  bool success = true;

  stringbuffer_truncate(output_string, 0);

  StringBuffer ship_name;
  stringbuffer_init(&ship_name);

  if (include_player)
  {
    success = get_shipname_from_type(&ship_name, graphics_set, ShipType_Player);
    if (success)
    {
      success = stringbuffer_append_all(
        output_string, stringbuffer_get_pointer(&ship_name));
    }
  }

  if (include_fighters)
  {
    for (ShipType i = ShipType_Fighter1; i <= ShipType_Fighter4 && success; i++)
    {
      stringbuffer_truncate(&ship_name, 0);
      success = get_shipname_from_type(&ship_name, graphics_set, i);
      if (success)
      {
        success = append_to_csv(
          output_string, stringbuffer_get_pointer(&ship_name));
      }
    }
  }

  stringbuffer_destroy(&ship_name);
  return success;
}
```

Direct append from an external source

Existing functions require the address and size of a buffer to be passed separately. Some functions can be modified to accept a `StringBuffer`, but that is not possible with third-party interfaces. In such cases, it is more efficient to expose the buffer wrapped by a `StringBuffer` for external writing than to allocate an intermediate buffer.

In the following example, the `messagetrans_lookup` function wraps a software interrupt instruction which calls an operating system routine [23]. The `msgsize` value output by `messagetrans_lookup` and passed into `stringbuffer_prepare_append` includes space for a null terminator, but the character count passed into `stringbuffer_finish_append` does not.

`msgsize` is recalculated by the second call to `messagetrans_lookup`, although the value is not expected to change. If the new value were bigger, it would indicate that `messagetrans_lookup` might have written outside the allocated buffer; if smaller, any excess space in the buffer would be kept for future append operations. The new value is used to update the string length.

`messagetrans_lookup` null-terminates its output, and `stringbuffer_finish_append` overwrites that terminator as if by `output_string[msgsize - 1] = '\0'`.

```
if (messagetrans_lookup(&messages, token,
                         NULL, 0, &msgsize, 1, id_string) == NULL)
{
  _Optional char *const outtail = stringbuffer_prepare_append(
                                     output_string, &msgsize);
  if (outtail &&
      messagetrans_lookup(&messages, token,
         outtail, msgsize, &msgsize, 1, id_string) == NULL)
  {
    stringbuffer_finish_append(output_string, msgsize - 1);
    return true;
  }
}
return stringbuffer_append_all(output_string, token);
```

Undo truncation for error handling

Undo can be used to handle run time errors by rolling back the state of a program. Use of a dedicated function avoids violating encapsulation or requiring a copy of the original string. The following code relies on `enter_dir` not to modify the truncated string.

```
/* Try to recreate the top-level data structure again. */
old_dir_list = iterator->dir_list;
linkedlist_init(&iterator->dir_list);
stringbuffer_truncate(&iterator->path_name,
                      iterator->path_name_len);

e = enter_dir(iterator);
if (e == NULL)
{
  /* Destroy the old data structures on success. */
  free_levels(&old_dir_list, NULL);
}
else
{
  /* Restore the previous state on error */
  iterator->dir_list = old_dir_list;
  stringbuffer_undo(&iterator->path_name);
}
```

Undo truncation to reinstate a leaf name

Undo isn't only useful for error handling, as in the following example where part of a string is used for a specific purpose before restoring the whole string.

```
/* Remove the leaf name of the current directory from the path */
stringbuffer_truncate(&iterator->path_name,
                      ancestor->path_name_len);

/* Try to refill the buffer with catalogue entries for the ancestor
   directory */
{
  DirIteratorLevel *tmp = ancestor;
  e = refill_buffer(iterator, &tmp);
  assert(tmp != NULL);
  ancestor = tmp;
}

/* Reinstate the leaf name of the current directory */
stringbuffer_undo(&iterator->path_name);
```

Undo an append to replace a leaf name

Undo is strictly redundant in the following example because the user could instead have stored the string's previous length and used `stringbuffer_truncate`.

```
/* Second append is a deliberate no-op to reset the undo state for the
   save path string buffer. */
if (!stringbuffer_append(&scan_data->save_path, ".", SIZE_MAX) ||
    !stringbuffer_append(&scan_data->save_path, NULL, 0))
{
  RPT_ERR("NoMem");
  return false;
}
```

And then later:

```
/* Remove the previous sub-path (does nothing if already undone) */
stringbuffer_undo(&scan_data->save_path);
e = append_to_string_buffer(&scan_data->save_path,
                            scan_data->iterator,
                            diriterator_get_object_sub_path_name);
```

GPU driver snippets

The following functions were developed for the Arm Mali GPU driver. They were implemented as a wrapper for a generic resizing array (instantiated for type `char`). The wrapper simply ensured null termination of the string and recorded the most recent run time error to occur.

Interface


```
typedef struct {
	int x;
} mali_error;

#define MALI_ERROR_NONE (mali_error){0};

bool mali_error_is_error(mali_error e)
{
	return e.x != 0;
}

typedef struct
{
	cutils_astring_array array;
	mali_error error;
} cutils_astring;

void cutils_astring_init(cutils_astring *astring /*out*/);

mali_error cutils_astring_ncat(cutils_astring *astring /*in,out*/,
                               char const *str /*in*/,
                               size_t n /*characters*/);

mali_error cutils_astring_cat(cutils_astring *astring /*in,out*/,
                              char const *str /*in*/);

mali_error cutils_astring_vprintf(cutils_astring *astring /*in,out*/,
                                  char const *format /*in*/,
                                  va_list args /*in*/);

mali_error cutils_astring_printf(cutils_astring *astring /*in,out*/,
                                 char const *format /*in*/,
                                 ...);

size_t cutils_astring_len(cutils_astring const *astring /*in*/);

char const *cutils_astring_ptr(cutils_astring const *astring /*in*/);

void cutils_astring_clear(cutils_astring *astring /*in,out*/);

mali_error cutils_astring_error(cutils_astring const *astring /*in*/);

void cutils_astring_term(cutils_astring *astring /*in */);
```

Description

The storage lifetime of the structure is under the client's control, which harms encapsulation but minimizes indirection and allows inline function definitions. The `cutils_astring_init` function cannot fail and is equivalent to default initialisation using `{}`. This could be turned into a useful guarantee.

All write functions append to the string in the buffer. This keeps the interface simple and prevents partial overwriting of multibyte characters.

Wide characters can be converted to multibyte characters and appended to the string by using the standard `l` length modifier:

```
wchar_t wstr[] = L"wide string";
cutils_astring_printf(astring, "%ls", wstr);
```

There is no support for:

- overwriting existing characters in a string.
- inserting characters and moving the tail of the string.
- direct insertion of strings into the buffer.
- truncation (except to length zero by calling `cutils_astring_clear`).
- undoing the last operation.

Additional storage is allocated automatically as the string grows. Functions that may allocate storage return an error indication which need not always be checked because `cutils_astring_error` can be called at any time to return the current error state. This allows lazy error handling when multiple strings are appended to the same buffer.

The user is responsible for calling `cutils_astring_term` to free any allocated storage. This is more type-safe than `free` but impedes interoperability with code that needs to take ownership of the underlying character array.

`cutils_astring_clear` clears any stored error as well as replacing the current string with the empty string. One rationale is that the stored string is no longer wrong; another is that ANSI C added the same behaviour to the standard `rewind` function [25].

Some functions are strictly redundant:

- `cutils_astring_cat(astring, tail)` is equivalent to `cutils_astring_ncat(astring, tail, SIZE_MAX)` or `cutils_astring_printf(astring, "%s", str)`.
- `cutils_astring_ncat(astring, tail, n)` is equivalent to `cutils_astring_printf(astring, "%.*s", n, str)`.

`cutils_astring_ptr` returns a pointer-to-`const` to prevent encapsulation violations. This prevents interoperability with functions that don't accept `const`-qualified strings but that hasn't been a problem yet.

Because the underlying string is accessible, there is no need for many overly specialized functions. For example, concatenation of two `cutils_astring` objects can be implemented as follows:

```
cutils_astring_cat(dest, cutils_astring_ptr(src));
```

POSIX

Two POSIX functions absent from ISO C have relevance to safe string handling:

- `fmemopen` associates a buffer of fixed size (which may be externally allocated) with an I/O stream [7].
- `open_memstream` creates an I/O stream associated with a buffer that it dynamically allocates internally [8].

The rationale is:

> *This interface has been introduced to eliminate many of the errors encountered in the construction of strings, notably overflowing of strings. This interface prevents overflow.*

According to Linux manuals, these functions first appeared in glibc 1.0.x (presumably in the mid-1990s). They first appeared in OpenBSD 5.4 (1 November 2013). They conform to IEEE Std 1003.1-2008 ("POSIX.1").

Interface


```
_Optional FILE *fmemopen(_Optional void *buf /*out*/,
                         size_t size,
                         const char *mode /*in*/);

_Optional FILE *open_memstream(char **bufp /*out*/, size_t *sizep /*out*/);

_Optional FILE *open_wmemstream(wchar_t **bufp /*out*/, size_t *sizep /*out*/);
```

Description

`FILE` need not be a complete type, but it is commonly defined as one. GCC 13.2.0 translates the following nonsensical code without producing a diagnostic:

```
FILE f;
fputc('c', &f);
```

cc65 2.19 defines `FILE` as an incomplete type, and therefore fails to translate the same program:

```
<source>:590: Error: Variable 'f' has unknown size
```

Writable streams opened by the POSIX functions are compatible with any function that accepts the address of a `FILE`. That minimizes the need for new functions but makes interfaces that operate on strings less recognizable. There is also nothing to prevent insertion of garbage using `fwrite`, so it's questionable whether streams are a type-safe interface to the underlying data.

By choosing the appropriate function and arguments, the programmer controls whether automatic or dynamic storage is used for the buffer, and whether the buffer size is fixed or variable. However, the names `fmemopen` and `open_memstream` are easily confused.

The array pointer and size passed to `fmemopen` are specified only once, which is safer than passing them in multiple calls to low-level string functions. However, they could still be wrong (just as for `snprintf`).

Usage of `fmemopen` is complex because it requires a `mode` argument and supposedly accepts all the same values as `fopen`, except when using an internally allocated buffer. This has been a source of bugs.

`fmemopen` allocates an internal buffer if called with a null pointer. Users cannot obtain a pointer to such a buffer.

Another feature of `fmemopen` is that it can append to a string already stored in the designated `buf` object. This mode requires the implementation to search for the first null byte in the buffer, which is used as the initial file position (if found). Consequently, predictable behaviour in “a” mode depends on the buffer having been pre-initialised.

Usage of `open_memstream` is simpler because it always allocates storage internally and the resultant stream is always writeable. However, at least one implementation treats the initial value of `*sizep` as a hint of the initial buffer size to allocate [9] (which is non-standard).

Write operations on a stream always overwrite existing characters at the current file position (which may be changed by calling `fsetpos` or `fseek`). A null character is only appended if characters are written beyond the previous string length. This isn’t immediately obvious so it may be surprising.

A stream opened by `open_memstream` is byte-oriented whereas `open_wmemstream` is wide-oriented. This implies that it is illegal to call functions such as `fwputs` on a stream opened by `open_memstream`, and illegal to call functions such as `fputs` on a stream opened by `open_wmemstream`. Such misuse is difficult to detect at compile time because *both functions return the same type of object*.

Streams opened by `open_wmemstream` have additional restrictions described in 7.23.2.5 of the C23 standard:

> *— Binary wide-oriented streams have the file-positioning restrictions ascribed to both text and binary streams.*
>
> *— For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide character output function can overwrite a partial multibyte character; any file contents beyond the byte(s) written may henceforth not consist of valid multibyte characters.*

This implies that `fseek` should only be used to seek the start of a string or a position previously returned by `ftell`; *it’s impossible to seek the end of a string*, to append to it. However, this is no guarantee that the integrity of multibyte characters will be maintained because a file position that was previously valid (including one stored by `fgetpos`) may be invalidated by overwriting at that location.

There is no support for:

- inserting characters and moving the tail of the string.
- direct insertion of strings into the buffer.
- truncation (except by writing a null character at the current file position).
- splitting the string.
- undoing the last operation.

Write functions (e.g., `fputs`, `fprintf`) return an error indication but also set the error indicator of the stream. This allows lazy error handling when multiple writes occur and only the final result is significant.

Writing more data than can fit in a buffer associated with a stream opened by `fmemopen` causes an error which may not be immediately visible because streams are buffered [10]. After this caused a bug report [11], the documentation was updated to suggest calling `setbuf(stream, NULL)` to disable buffering.

Writing more data than can fit in a buffer associated with a stream opened by `open_memstream` causes additional storage to be allocated. The user is responsible for calling `free` to deallocate the buffer when no longer required. Because no unique type is defined, it might not be obvious which strings in a program should be freed and which must not. However, this facilitates interoperability with code that needs to take ownership of the buffer.

A stream created by either function must be closed by calling `fclose`, which must be called before freeing the buffer allocated by `open_memstream`. Closing a stream opened by `fmemopen` automatically frees the associated buffer, if internal.

A null character is automatically written at the end of the buffer when a stream opened by either function is flushed or closed. Moreover, null is only written to a buffer associated with a stream opened by `fmemopen` *if there is space*. This is a potential vulnerability.

Although state associated with the stream is encapsulated, the buffer can still be a source of programmer errors:

- The `buf` object designated by a call to `fmemopen` is fully accessible but may not contain a null character unless `fflush` or `fclose` has just been called.
- The `*buf` and `*sizep` objects designated by a call to `open_memstream` are assigned a value whenever `fflush` is called, at which point `*buf` points to the buffer and `*sizep` is its length (or the current file position, if lower).
- Previous values of `*buf` and `*sizep` are invalidated by a write operation or a call to `fclose` so the user must be careful not to use stale values.

One significant advantage the POSIX functions have is that it is impossible to obtain access to the underlying buffer from the address of the associated `FILE` alone. Hence, it is possible to pass `FILE *` to an untrusted function but withhold direct access to the buffer.

However, streams have complex behaviour which isn't always appropriate. Some users may be put off by the fact that `fputs` is perceived to be costly whereas `strcat` is perceived to be lightweight, even though the latter performs a linear search.

GNU/BSD

Two BSD functions [12] absent from both ISO C and POSIX have relevance to safe string handling:

- `asprintf` allocates storage for a string generated under control of a format string, as a single operation.
- `vasprintf` is equivalent to `asprintf` except that it accepts a single argument of type `va_list` in place of arguments to be substituted for format specifiers.

These functions are declared by `<stdio.h>` if `_GNU_SOURCE` is defined.

They first appeared in the GNU C library before February 1995. They were later added to FreeBSD 2.2 (March 1997) and OpenBSD 2.3 (May 1998). They were added to Oracle Solaris 10 8/11 (Update 10) in 2011.

Interface

Copyright 1994-2024 The FreeBSD Project.

```
int asprintf(_Optional char **strp /*out*/, const char *format /*in*/, ...);

int vasprintf(_Optional char **strp /*out*/,
              const char *format /*in*/,
              va_list ap /*in*/);
```

Description

These functions prevent buffer overruns since their result is dynamically allocated. However, they introduce new potential errors: neglecting to check that storage allocation succeeded or neglecting to free the result string.

The return value indicates failure (-1), or the number of characters generated (excluding the terminating null). The caller must always check the return value because the Linux manual [13] says that the designated `*strp` pointer is undefined on failure, whereas the Free BSD manual says that it is set to null. This would need to be cleared up as part of any attempt at standardisation.

This interface resembles `sprintf` rather than `strdup`. Unfortunately, that makes it awkward and error-prone to call one of these functions as part of a variable declaration:

```
_Optional char *s = asprintf(&s, "%d", 99) >= 0 ? s : NULL;
```

Consequently, `s` cannot be declared as immutable and is likely to be declared with a wider scope than required – either as an uninitialised object or initialised to a value that is never used.

Every alternative version (including similarly named Linux kernel functions) seems to instead return a pointer to the allocated storage, or null on failure. This suggests that the returned string length is rarely wanted. It might also help to explain why the GNU/BSD functions were never standardized, even though many programs require equivalent functionality.

Storage is allocated as if by calling `malloc` and the user is responsible for calling `free` to deallocate it when no longer required. Because no unique type is defined, it might not be obvious which strings in a program should be freed and which must not. However, this facilitates interoperability with code that needs to take ownership of the string.

Like a string allocated by `open_memstream` or `strdup`, the result string is guaranteed to be terminated by a null character.

Linux kernel

Several kernel functions [14] have relevance to safe string handling:

- `kasprintf` allocates storage for a string generated under control of a format string, as a single operation.
- `kvasprintf` is equivalent to `kasprintf` except that it accepts a single argument of type `va_list` in place of arguments to be substituted for format specifiers.

These functions are declared in `<linux/sprintf.h>`.

The `kasprintf` function was added to the Linux kernel on Jun 25, 2006 [15] and `kvasprintf` on May 1, 2007.

Interface

Copyright (C) 1991, 1992 Linus Torvalds.

```
_Optional char *kasprintf(gfp_t gfp /*in*/, const char *fmt /*in*/, ...);

_Optional char *kvasprintf(gfp_t gfp /*in*/,
                           const char *fmt /*in*/,
                           va_list args /*in*/);
```

Description

This interface is designed to resemble `strdup` rather than `sprintf`. This makes it easy to call one of these functions as part of a variable declaration:

```
_Optional char *const s = kasprintf(GFP_KERNEL, "%d", 99);
```

Consequently, `s` can be declared as immutable if appropriate and it is less tempting to declare it as an uninitialised variable.

Apart from this simplification, the Linux kernel functions behave like `asprintf` and `vasprintf` with all the same advantages and drawbacks.

GLib

This library was originally part of GTK (GIMP Toolkit) [16]. It is now a standalone general-purpose library which provides data structures and other utilities, including a managed string type: `GString` [17].

The rationale is:

> *Crucially, the "str" member of a GString is guaranteed to have a trailing nul character, and it is therefore always safe to call functions such as strchr() or strdup() on it.*

The string functions were already present in GLib 1.1.12 (Jan 4, 1999) [18] but were probably introduced earlier.

Interface

Copyright (C) 1995-1997 Peter Mattis, Spencer Kimball and Josh MacDonald.

```
typedef struct _GString GString;

struct _GString
{
  gchar *str;
  gsize len;
  gsize allocated_len;
};

GString *g_string_new(_Optional const gchar *init);

GString *g_string_new_len(_Optional const gchar *init,
                          gssize len);

GString *g_string_sized_new(gsize dfl_size);

gchar *g_string_free(GString *string, gboolean free_segment);
gchar *g_string_free_and_steal(GString *string);
GBytes *g_string_free_to_bytes(GString *string);

gboolean g_string_equal(const GString *v, const GString *v2);

guint g_string_hash(const GString *str);

GString *g_string_assign(GString *string, const gchar *rval);

GString *g_string_truncate(GString *string, gsize len);


GString *g_string_set_size(GString*string, gsize len);

GString *g_string_insert_len(GString *string,
                             gssize pos,
                             const gchar *val,
                             gssize len);

GString *g_string_append(GString *string,
                         const gchar *val);

GString *g_string_append_len(GString *string,
                             const gchar *val,
                             gssize len);

GString *g_string_append_c(GString *string,
                           gchar c);
```

```
GString *g_string_append_unichar(GString *string,
                                 gunichar wc);

GString *g_string_prepend(GString *string,
                          const gchar *val);

GString *g_string_prepend_c(GString *string,
                            gchar c);

GString *g_string_prepend_unichar(GString *string,
                                  gunichar wc);

GString *g_string_prepend_len(GString *string,
                              const gchar *val,
                              gssize len);

GString *g_string_insert(GString *string,
                         gssize pos,
                         const gchar *val);

GString *g_string_insert_c(GString *string,
                           gssize pos,
                           gchar c);

GString *g_string_insert_unichar(GString *string,
                                 gssize pos,
                                 gunichar wc);

GString *g_string_overwrite(GString *string,
                            gsize pos,
                            const gchar *val);

GString *g_string_overwrite_len(GString *string,
                                gsize pos,
                                const gchar *val,
                                gssize len);

GString *g_string_erase(GString *string,
                        gssize pos,
                        gssize len);

guint g_string_replace(GString *string,
                       const gchar *find,
                       const gchar *replace,
                       guint limit);

GString *g_string_ascii_down(GString *string);
GString *g_string_ascii_up(GString *string);

void g_string_printf(GString *string,
                     const gchar *format,
                     ...);

void g_string_append_printf(GString *string,
                            const gchar *format,
                            ...);
```

Description

`GString` is defined as a complete type, which harms encapsulation. It's possible to declare an object of that type but unclear what such a declaration would mean. `GString` objects are always accessed via a pointer, therefore an extra level of indirection is needed. However, use of a complete type does allow inline functions such as `g_string_append_len` and `g_string_append_c`.

`g_string_new` allocates storage for a `GString` object and returns its address. It also allocates storage for the underlying character array. At one time, memory was allocated from a fast slab allocator with per-thread free lists; since GLib 2.76, `malloc` is used instead. Allocating storage separately for the `GString` and its character array has overheads that are apparently not considered significant.

If any attempt at storage allocation fails, then *the program is terminated*. This would be unacceptable for an interactive application running on a machine with limited physical memory and no swap file. A question on Stack Overflow suggests that not all users are satisfied with it [19].

If the `init` parameter is not null, then it points to a null terminated string to be copied as the initial value of the underlying character array; otherwise, storage is allocated for an empty string. The `g_string_new_len` variant limits the number of characters copied, whereas `g_string_sized_new` allows the user to specify the amount of storage to be pre-allocated. These could easily be confused.

Functions are provided to overwrite or insert into the buffer at arbitrary positions, in addition to the more common operation of appending. This results in a larger interface which is harder to learn, and a more complex implementation. It also has implications for multibyte characters.

Arguably, the only thing that a function called by a character producer ought to do is consume characters. Requiring extra parameters risks pushing decisions about whether to insert or overwrite, and where to do it, further down the call stack into functions that could otherwise be used as general-purpose producers. This is harmful to composability.

Unicode characters are converted to multibyte characters upon insertion into a `GString`. These are written one at a time (as if by `fputwc`). Unlike a stream, there is no notion of a `GString` becoming 'wide-oriented' or otherwise special because of a call to a function like `g_string_insert_unichar`.

Wide characters (typically UTF-16) can be converted to multibyte characters and appended to the string by using the standard `l` length modifier:

```
wchar_t wstr[] = L"wide string";
g_string_append_printf(string, "%ls", wstr);
```

Many functions have a position parameter to allow random access to the array (e.g., `g_string_insert`). This is interpreted as a byte offset rather than a character offset, therefore any call to those functions has the potential to partially overwrite the end of a multibyte character. Functions that overwrite or delete characters instead of inserting (e.g., `g_string_overwrite` or `g_string_erase`) can cause similar errors at the end of the affected region.

Additional storage for the underlying character array is allocated automatically as the string grows, for example because of calls to `g_string_append`. This can terminate the program on failure.

The user is responsible for calling `g_string_free` to free a `GString` object. Optionally, the underlying character array is also automatically freed; passing false as the value of `free_segment` or using an alternative function, `g_string_free_and_steal`, prevents that. This facilitates interoperability with code that needs to take ownership of a string.

Almost every function returns the address of a `GString` object, which has no use other than to facilitate nested function calls since it is always the same as the address passed by the caller.

This allows idiomatic use of `GString` functions resembling use of `strcpy` and `strcat`:

```
GString *dst = g_string_new("foo");
g_string_append(g_string_assign(dst, "bar"), "qux");
```

Is analogous to:

```
char dst[12] = "foo";
strcat(strcpy(dst, "bar"), "qux");
```

However, this feature may be confusing since it is unclear whether a function's return value can safely be ignored. Returning redundant values also has a cost, albeit tiny.

Some position and length parameters use a signed type, `gssize`, instead of the equivalent unsigned type `gsize`. This is not entirely consistent (e.g., `g_string_erase` doesn't allow `pos < 0`, despite appearances) but usually:

- `pos < 0` means insert at end of string.
- `len < 0` means insert/erase all characters to end of string.

There isn't a precedent for this in the standard since the `size_t` and `rsize_t` types used for character counts are unsigned, and `ssize_t` is non-standard.

Use of in-band error indicators is deprecated by many authors [20]. In-band special argument values can cause similar hazards.

For example, if an argument is negative because of accidentally swapped operands in a pointer subtraction, it may cause unexpected behaviour:

```
const char *names = "james;joyce;nancy;wilfred;",
           *start = names;
for (_Optional const char *end = strchr(start, ';');
     end != NULL;
     start = end + 1, end = strchr(start, ';'))
{
    GString *s = g_string_new_len(start, start - end); // whoops!
}
```

(GLib can be configured to report out-of-range argument values, but the values accidentally passed above are not out of range.)

Many functions are strictly redundant, but the alternatives are often cryptic:

- `g_string_new(init)` is equivalent to `g_string_new_len(init, -1)`.
- `g_string_prepend(string, val)` is equivalent to `g_string_insert_len(string, 0, val, -1)`.
- `g_string_prepend_len(string, val, len)` is equivalent to `g_string_insert_len(string, 0, val, len)`.
- `g_string_insert(string, pos, val)` is equivalent to `g_string_insert_len(string, pos, val, -1)`.
- `g_string_append(string, val)` is equivalent to `g_string_insert_len(string, -1, val, -1)`.
- `g_string_append_len(string, val, len)` is equivalent to `g_string_insert_len(string, -1, val, len)` or `g_string_append_printf(string, "%.*s", len, val)`.
- `g_string_overwrite(string, pos, val)` is equivalent to `g_string_overwrite_len(string, pos, val, -1)`.
- `g_string_prepend_c(string, c)` is equivalent to `g_string_insert_c(string, 0, c)`.
- `g_string_append_c(string, c)` is equivalent to `g_string_insert_c(string, -1, c)` or `g_string_append_printf(string, “%c”, c)`.
- `g_string_prepend_unichar(string, wc)` is equivalent to `g_string_insert_unichar(string, 0, wc)`.
- `g_string_append_unichar(string, wc)` is equivalent to `g_string_insert_unichar(string, -1, wc)`.
- `g_string_free_and_steal(string)` is equivalent to `g_string_free(string, FALSE)`.
- `g_string_assign(string, rval)` is equivalent to `g_string_printf(string, "%s", rval)`.
- `g_string_append(string, val)` is equivalent to `g_string_append_printf(string, "%s", val)`.
- `g_string_truncate(string, len)` is equivalent to `g_string_erase(string, len, -1)`.

Function popularity

The following data was collected from GitHub code search results [24]. Larger values were reported to the nearest hundred. The table is sorted by the number of files each function was reported to appear in.

Appending and truncation appear to be the most widely used operations. Pre-allocation of a specified buffer size is surprisingly popular. Replacement of a whole string is moderately common. Inserting into a string (including prepending) is relatively uncommon, and overwriting is the least common operation of all.

Function	No. of files
`g_string_free`	64500
`g_string_new`	61400
`g_string_append`	41700
`g_string_append_printf`	37800
`g_string_append_c`	25500
`g_string_sized_new`	13900
`g_string_truncate`	12200
`g_string_append_len`	11300
`g_string_printf`	6600
`g_string_assign`	5800
`g_string_erase`	4400
`g_string_set_size`	4300
`g_string_new_len`	3200
`g_string_append_unichar`	2400
`g_string_prepend`	2400
`g_string_insert`	1800
`g_string_prepend_c`	1300
`g_string_insert_c`	936
`g_string_equal`	764
`g_string_insert_len`	600
`g_string_free_to_bytes`	528
`g_string_hash`	436
`g_string_prepend_len`	434
`g_string_ascii_down`	406
`g_string_replace`	328
`g_string_insert_unichar`	318
`g_string_overwrite_len`	290
`g_string_ascii_up`	280
`g_string_prepend_unichar`	261
`g_string_overwrite`	257
`g_string_free_and_steal`	109

ImageMagick

ImageMagick is an open-source software suite for editing digital images, with a particular emphasis on scripting. It was first released on August 1st, 1990 [21]. Its 'MagickCore string methods' are part of the publicly available source code [22]. They are copyright 1999 ImageMagick Studio LLC. A subset is reproduced here for illustrative purposes.

Interface


```
typedef struct _StringInfo
{
  _Optional char *path;
  _Optional unsigned char *datum;
  size_t length, signature;
  _Optional char *name;
} StringInfo;

_Optional char *StringInfoToString(const StringInfo *string_info);
int CompareStringInfo(const StringInfo *target, const StringInfo *source);
size_t GetStringInfoLength(const StringInfo *string_info);

StringInfo
  *AcquireStringInfo(size_t length),
  *CloneStringInfo(const StringInfo *string_info),
  *DestroyStringInfo(StringInfo *string_info),
  *SplitStringInfo(StringInfo *string_info, size_t offset),
  *StringToStringInfo(const char *string);

_Optional StringInfo *BlobToStringInfo(_Optional const void *blob, size_t length);
_Optional unsigned char *GetStringInfoDatum(const StringInfo *string_info);

void
  ConcatenateStringInfo(StringInfo *string_info, const StringInfo *source),
  ResetStringInfo(StringInfo *string_info),
  SetStringInfo(StringInfo *string_info,const StringInfo *source),
  SetStringInfoDatum(StringInfo *string_info, const unsigned char * source),
  SetStringInfoLength(StringInfo *string_info, size_t length);
```

Description

`StringInfo` is defined as a complete type, which harms encapsulation. It's possible to declare an object of that type but that is unlikely to yield a valid object since every function asserts `string_info->signature == MagickCoreSignature`. There's no need for `StringInfo` to be a complete type because not even trivial operations such as `GetStringInfoLength` have inline function definitions.

`AcquireStringInfo` allocates storage for a `StringInfo` object and returns its address. It also allocates a character array of the requested length plus 4KB and zero-initialises it. By default, `malloc` is used. If storage cannot be allocated, then *the program is terminated*. Allocating memory separately for the `StringInfo` and character array has overheads that are apparently not considered significant.

`BlobToStringInfo` is a variant of `AcquireStringInfo` which allows the caller to pass the address of a string to be copied. `BlobToStringInfo(NULL, length)` is equivalent to `AcquireStringInfo(length)`. It isn't significant whether the source string contains a null character or not. This allows a `StringInfo` object to be constructed from a substring. The first null character may occur earlier than implied by the stored length.

`StringToStringInfo` is a variant of `AcquireStringInfo` that accepts a null terminated string to be copied as the initial value of the underlying character array. Other, highly specialized, constructors also exist.

The string initially stored in a `StringInfo` object is null terminated since the underlying array is over-allocated and padded with zeros. It's unclear whether that is intentional, since other functions ensure there is always 4KB extra space but neglect to initialise any new storage acquired beyond the string.

Additional storage for the underlying character array is allocated automatically as the string grows, for example because of calls to `ConcatenateStringInfo`. This can terminate the program on failure.

Functions are provided to truncate or replace the string in the buffer, append the content of another string buffer, or split it into separately allocated buffers.

There is no support for:

- inserting characters and moving the tail of the string.
- appending characters generated under control of a format string.
- appending characters from a character array.
- undoing the last operation.
- truncating a string without reducing the size of the underlying array.

`GetStringInfoDatum` returns an unqualified pointer to the first character of the string. This allows interoperability with functions which don't accept `const`-qualified strings, or which temporarily modify a passed-in string. Arguably, it violates encapsulation. The exposed string isn't necessarily null terminated.

`GetStringInfoLength` is typically used with `GetStringInfoDatum`, for example to efficiently access characters indexed relative to the end of the string:

```
exif_length=GetStringInfoLength(exif_profile);
exif_datum=GetStringInfoDatum(exif_profile);
if ((exif_length > 2) &&
    ((memcmp(exif_datum,"\xff\xd8",2) == 0) ||
     (memcmp(exif_datum,"\xff\xe1",2) == 0)) &&
    (memcmp(exif_datum+exif_length-2,"\xff\xd9",2) == 0))
  SetStringInfoLength(exif_profile,exif_length-2);
```

`SetStringInfoLength` sets the length stored in a `StringInfo` object and increases or reduces the amount of storage allocated for its underlying character array accordingly. By default, `realloc` is used. However, *no null character is written or removed*. Consequently, the first null character in the underlying array (if any) may be earlier or later than implied by the new length.

`SetStringInfoDatum` copies as many characters into a `StringInfo` object as its stored length permits, truncating the `source` data if necessary to fit. It isn't significant whether `source` contains a null character, and *no null is appended* to the result. Afterwards, the destination array may not contain null. `SetStringInfo` is similar but may be safer because it doesn't copy more characters than the source `StringInfo` provides.

Often, length and datum are set consecutively:

```
SetStringInfoLength(nonce,sizeof(extent));
SetStringInfoDatum(nonce,(const unsigned char *) &extent);
```

ConcatenateStringInfo increases the storage allocated for one StringInfo to concatenate the content of another. It copies all characters from the source object and stores the sum of both lengths. It isn't significant whether source contains a null character, and *no null is appended* to the result. Afterwards, the destination array may not contain null.

SplitStringInfo constructs a new StringInfo by copying the first offset characters of an existing StringInfo object. If successful, it moves the remaining characters of the underlying array to the start, reduces its stored length and shrinks the allocated storage.

CloneStringInfo constructs a new StringInfo by copying one more than the number of characters indicated by the stored length of an existing StringInfo object. It isn't significant whether the array contains a null terminator. The clone has the same stored length as the original.

StringInfoToString allocates storage for a copy of the array underlying a StringInfo object, copies the number of characters indicated by its stored length, then appends a null terminator (like strndup). The caller takes ownership of the newly allocated array.

Handling of storage allocation failure is inconsistent:

- StringInfoToString always returns NULL on failure.
- BlobToStringInfo may return NULL or terminate (depending on where it fails).
- AcquireStringInfo, StringToStringInfo, CloneStringInfo, SetStringInfoLength and ConcatenateStringInfo always terminate on failure.

CompareStringInfo is equivalent to strcmp except that it limits the number of characters compared to the smaller of the lengths stored in the two StringInfo objects instead of relying on null termination.

ResetStringInfo sets all characters of the underlying array to null without altering the stored length. It's unclear what purpose this was intended to serve, since all extant calls are redundant:

```
  random_info->nonce=AcquireStringInfo(2*GetSignatureDigestsize(
    random_info->signature_info));
  ResetStringInfo(random_info->nonce);
  random_info->reservoir=AcquireStringInfo(GetSignatureDigestsize(
    random_info->signature_info));
  ResetStringInfo(random_info->reservoir);
```

The user is responsible for calling DestroyStringInfo to free a StringInfo object and its underlying character array. This is more type-safe than free but impedes interoperability with code that needs to take ownership of the array.

DestroyStringInfo always returns NULL. This is typically assigned to the same variable as the input argument, presumably as an idiomatic way of reducing the likelihood of dangling pointers:

```
status=SetImageProfile(image,name,profile,exception);
profile=DestroyStringInfo(profile);
```

Such an assignment may be omitted if the next statement assigns a value to the same variable:

```
(void) SetImageProfile(image,"EXIF",profile);
DestroyStringInfo(profile);
profile=GetImageProfile(image,"EXIF");
```

Consequently, the compiler cannot warn about the return value of DestroyStringInfo being ignored (e.g., using -Wunused-result).

Comparison of usage

A simple program has been implemented using each of the interfaces previously discussed. It converts an array of integer indices into a comma-separated list of names. To make this a fair comparison, the program does not exercise features missing from most of interfaces (e.g., prepending or insertion).

Standard C functions

When increasing the amount of storage allocated for the array, its size is doubled if that is sufficient to store the new string. This reduces the number of reallocation operations. Enough space is allocated for every name in the list to be preceded by a comma even though this is excessive for the first name.

For simplicity, `strcat` is used to append strings instead of treating the current length as another variable and using `memcpy` (which would be more efficient). The stored string is truncated by writing a null character at the end of each iteration of the outer loop, which allows same array to be reused for the next iteration.

It's tempting to refactor this code into two functions: one to contain the business logic and the other to encapsulate the complexity of appending a string to the array. That would lead to something resembling several of the other solutions.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))

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;
}
```

CBUtilLib

It’s worth reusing the same `StringBuffer` object between iterations of the outer loop because `stringbuffer_truncate` truncates the stored string without minimizing its allocated size. Strings can be appended without first copying them into `StringBuffer` objects.

If an allocation failure has occurred, then no more strings are appended. However, the program is responsible for maintaining the error indicator correctly to ensure this.

```
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}};
    StringBuffer buf;
    stringbuffer_init(&buf);
    bool ok = true;

    for (size_t i = 0; i < ARRAY_SIZE(data); ++i) {
        for (size_t j = 0; j < ARRAY_SIZE(data[0]); ++j) {
            if (j > 0 && ok)
                ok = stringbuffer_append_all(&buf, ",");

            if (ok)
                ok = stringbuffer_append_all(&buf, names[data[i][j]]);
        }

        if (!ok) {
            fprintf(stderr, "Failed at %zu (length %zu)\n",
                    i, stringbuffer_get_length(&buf));
            break;
        }

        puts(stringbuffer_get_pointer(&buf));
        stringbuffer_truncate(&buf, 0);
    }

    stringbuffer_destroy(&buf);
    return ok ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

GPU driver snippets

It's worth reusing the same `cutils_astring` object between iterations of the outer loop because `cutils_astring_clear` truncates the stored string without minimizing its allocated size. Strings can be appended without first copying them into `cutils_astring` objects.

Instead of checking for failure of each attempted concatenation, the error indicator is checked once per iteration of the outer loop. It must be checked before calling `cutils_astring_clear`, which resets it.

```
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}};
    cutils_astring buf;
    cutils_astring_init(&buf);

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

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

        if (mali_error_is_error(cutils_astring_error(&buf))) {
            fprintf(stderr, "Failed at %zu (length %zu)\n",
                    i, cutils_astring_len(&buf));
            break;
        }

        puts(cutils_astring_ptr(&buf));
        cutils_astring_clear(&buf); // also clears any error
    }

    int err = mali_error_is_error(cutils_astring_error(&buf)) ?
                EXIT_FAILURE : EXIT_SUCCESS;
    cutils_astring_term(&buf);
    return err;
}
```

POSIX

The buffer allocated by `open_memstream` is reused by calling `rewind` between iterations of the outer loop. This also clears the stream's error indicator but does not truncate the string or change the buffer's allocated size.

Calls to `fputs` overwrite characters at the current file position without necessarily appending a null character, therefore a call to `fputc` is required to ensure that the string is terminated in the correct place during the second iteration and subsequent iterations of the outer loop.

Neither the buffer's address and size, nor the stream's error indicator, are visible until the call to `fflush`. Instead of checking for failure of each attempted put, the error indicator is checked once per iteration of the outer loop.

```
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}};
    char *buf;
    size_t buf_size;
    _Optional FILE *f = open_memstream(&buf, &buf_size);
    if (f == 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)
                fputs(",", f);
            fputs(names[data[i][j]], f);
        }
        fputc('\0', f);
        fflush(f);

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

        puts(buf);
        rewind(f); // also clears any error
    }

    int err = ferror(f) ? EXIT_FAILURE : EXIT_SUCCESS;
    fclose(f);
    free(buf);
    return err;
}
```

GNU/BSD

Each iteration of the inner loop allocates a new character array which contains a concatenation of the string generated so far with the next name in the list, after which the previous array is freed. This is inefficient in terms of the number of storage allocation requests and number of bytes copied, but the actual amount of storage used is minimized.

Errors are handled immediately, and the complexity of doing this correctly is non-trivial.

```
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;
    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) {
            _Optional char *new_buf;
            int n = asprintf(
                        &new_buf,
                        "%s%s%s",
                        buf ? buf : "",
                        j > 0 ? "," : "",
                        names[data[i][j]]);

            if (n < 0)
            {
                err = EXIT_FAILURE;
                break;
            }
            free(buf);
            buf = new_buf;
        }

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

        puts(buf);
        strcpy(buf, "");
    }

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

Linux kernel

This is like the GNU/BSD code but uses a simpler function to append each name in the list.

```
int print_fruit(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;
    int err = 0;

    for (size_t i = 0; i < ARRAY_SIZE(data); ++i) {
        for (size_t j = 0; j < ARRAY_SIZE(data[0]); ++j) {
            _Optional char *new_buf = kasprintf(
               GFP_KERNEL,
               "%s%s%s",
               buf ? buf : "",
               j > 0 ? "," : "",
               names[data[i][j]]);

            if (new_buf == NULL)
            {
                err = -ENOMEM;
                break;
            }
            kfree(buf);
            buf = new_buf;
        }

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

        printk(KERN_INFO "%s\n", buf);
        strcpy(buf, "");
    }

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

GLib

It's worth reusing the same `GString` object between iterations of the outer loop because `g_string_truncate` truncates the stored string without minimizing its allocated size. Strings can be appended without first copying them into `GString` objects.

No errors are handled because execution terminates if storage allocation fails.

```
int main(void)
{
    gchar const *const names[] = {"apple", "orange", "banana", "lime"};
    size_t const data[][6] = {{3,0,2,0,1,0}, {1,2,0,3,3,0}};
    GString *buf = g_string_new(NULL);

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

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

        puts(buf->str);
        buf = g_string_truncate(buf, 0);
    }

    (void)g_string_free(buf, TRUE);
    return EXIT_SUCCESS;
}
```

ImageMagick

It's impossible to append a string without copying it into a `StringInfo`, so a single copy of the comma string is created and reused. A `StringInfo` object is not reused between iterations of the outer loop because truncating the stored string would also minimize its allocated size. This is the least efficient code, whether measured in terms of number of storage allocation requests, amount of storage allocated, or number of bytes copied.

No errors are handled because execution terminates if storage allocation fails.

```
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}};
    StringInfo *comma = StringToStringInfo(",");

    for (size_t i = 0; i < ARRAY_SIZE(data); ++i) {
        StringInfo *buf = AcquireStringInfo(0);
        for (size_t j = 0; j < ARRAY_SIZE(data[0]); ++j) {
            if (j > 0)
                ConcatenateStringInfo(buf, comma);

            StringInfo *name = StringToStringInfo(names[data[i][j]]);
            ConcatenateStringInfo(buf, name);
            name = DestroyStringInfo(name);
        }

        printf("%.*s\n", (int)GetStringInfoLength(buf), GetStringInfoDatum(buf));
        buf = DestroyStringInfo(buf);
    }

    comma = DestroyStringInfo(comma);
    return EXIT_SUCCESS;
}
```

Lessons learned

- A string buffer should be specified using a single parameter.
- Storage allocation for strings should be automatic.
- Recalculation of free space in a buffer should be automatic.
- The effect of ignoring return values should be minimized.
- Where error checks are required, they should be simple.
- Aspects of character consumption should not be delegated to character producers.
- Appending and whole string replacement must be supported. Inserting into a buffer is uncommon, and overwriting is the least common operation of all.
- Restrictions on function usage should be expressed through types.
- Random access to multibyte characters is universally unsafe but often permitted.
- A surprising number of users want to control pre-allocation of storage.
- Direct insertion is tricky because of null termination.

Why not standardize existing functions?

Existing string-handling functions all have different pros and cons.

The POSIX functions, which most closely resemble my proposed interface have:

- An extra object whose lifetime must be managed correctly with respect to the buffer.
- Surprising behaviour related to unwanted internal 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.

The GLib functions, which seem to be popular now, have:

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

Nevertheless, I believe that standardizing any of the existing interfaces would be preferable to doing nothing, if the committee believes that is as far as their purview extends.

Proposed new functions

Interface

Core

Functions to be implemented by all conforming implementations.

```
typedef struct strb_t strb_t;

typedef struct { ... } strbstate_t;

strb_t *strb_use(strbstate_t *state /*out*/,
                 size_t size /*incl. null*/,
                 char buf[size] /*in*/);

_Optional strb_t *strb_reuse(strbstate_t *state /*out*/,
                             size_t size /*incl. null*/,
                             char buf[size] /*in*/);

const char *strb_ptr(strb_t const *sb /*in*/);

size_t strb_len(strb_t const *sb /*in*/);

enum {
  strb_insert,
  strb_overwrite
};

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

int strb_seek(strb_t *sb /*in*/, size_t pos);
size_t strb_tell(strb_t const *sb /*in*/);

bool strb_error(strb_t const *sb /*in*/);
void strb_clearerr(strb_t *sb /*in,out*/);

int strb_putc(strb_t *sb /*in,out*/, int c);

int strb_nputc(strb_t *sb /*in,out*/,
               int c,
               size_t n /*characters*/);

int strb_unputc(strb_t *sb /*in,out*/);

int strb_puts(strb_t *sb /*in,out*/, char *str /*in*/);

int strb_nputs(strb_t *sb /*in,out*/,
               const char *str /*in*/,
               size_t n /*characters*/);

_Optional char *strb_write(strb_t *sb /*in,out*/,
                           size_t n /*characters*/);

void strb_wrote(strb_t *sb /*in,out*/);

void strb_delto(strb_t *sb /*in,out*/, size_t pos);

int strb_cpy(strb_t *sb /*in,out*/, const char *str /*in*/);

int strb_ncpy(strb_t *sb /*in,out*/,
              const char *str /*in*/,
              size_t n /*characters*/);
```

Extended

Functions to be implemented by conforming hosted implementations.

```
_Optional strb_t *strb_alloc(size_t size /*incl. null*/);

_Optional strb_t *strb_dup(const char *str /*in*/);
_Optional strb_t *strb_ndup(const char *str /*in*/, size_t n /*characters*/);

_Optional strb_t *strb_aprintf(const char *format /*in*/, ...);
_Optional strb_t *strb_vaprintf(const char *format /*in*/,
                                va_list args /*in*/);

void strb_free(_Optional strb_t *sb /*in*/);

int strb_vputf(strb_t *sb /*in,out*/,
               const char *format /*in*/,
               va_list args /*in*/);

int strb_putf(strb_t *sb /*in,out*/,
              const char *format /*in*/,
              ...);

int strb_vprintf(strb_t *sb /*in,out*/,
                 const char *format /*in*/,
                 va_list args /*in*/);

int strb_printf(strb_t *sb /*in,out*/,
                const char *format /*in*/,
                ...);
```

Description

Object types

strbstate_t is an object type capable of recording all the information needed to control a string buffer, including a position indicator, the buffer's size, the length of the string in the buffer, and an error indicator. It is a complete type so it can be used for storage allocation, but its members are unspecified.

strb_t is an object type to be used in place of strbstate_t for all operations on a string buffer. It need not be a complete type. Consequently, it may be impossible to wrongly declare an object of type strb_t and its internal state should only be accessed by the provided functions.

An object of type strbstate_t can only be used to construct strb_t objects. Its storage duration must exceed the program's usage of any strb_t pointers derived from it. The effect on any derived strb_t object of modifying the originating strbstate_t object is undefined.

External buffers

Two functions are specified to create a strb_t object for the purpose of managing a character array:

- strb_use
- strb_reuse

The above functions initialise a strb_t object and return its address. Their caller must pass a strbstate_t object to be used to store information about the buffer.

The buf parameter to strb_use specifies a character array to be used instead of an internal buffer, and size specifies its size (which must not be 0). A null character is written as the first character. The initial string length is 0.

The array passed into strb_reuse should contain at least one null character within the first size characters, whose position is the initial string length. Any following characters are ignored. If no null character is found within the first size characters (or the maximum supported length, if less), a null pointer is returned.

Internal buffers

Various functions are specified to create a strb_t object that manages an internal buffer:

- strb_alloc
- strb_dup
- strb_ndup
- strb_aprintf
- strb_vaprintf

The above functions allocate storage for a strb_t object, initialise it, and return its address. They do not necessarily allocate a character array if the initial string is empty, providing that all other functions operate as if an array had been allocated. If storage allocation fails, a null pointer is returned.

The size parameter to strb_alloc is a hint about how much storage to allocate for an internal buffer (where 0 means default size). The initial string length is 0.

The buf parameter to strb_use specifies a character array to be used instead of an internal buffer, and size specifies its size (which must not be 0). A null character is written as the first character. The initial string length is 0.

The array passed into `strb_reuse` should contain at least one null character within the first `size` characters, whose position is the initial string length. Any following characters are ignored. If no null character is found within the first `size` characters (or the maximum supported length, if less), a null pointer is returned.

The effects of `strb_`... functions on an external array are always immediately visible and the string therein is always null terminated.

The `str` parameter to `strb_dup` points to a null terminated string to be copied as the initial value of the buffer associated with a `strb_t` object. The `strb_ndup` variant limits the number of characters copied.

The `format` parameter to `strb_printf` and `strb_vprintf` specifies how to convert subsequent arguments to generate a string, which forms the initial value of the buffer associated with a `strb_t` object.

The user is responsible for calling `strb_free` to free a `strb_t` object. The associated buffer is also automatically freed, except in the case where its address was passed as a parameter to `strb_use` or `strb_reuse`. This must be enforced because storage allocation is abstracted. To pass an internally allocated string to code that needs to take ownership of it, it must first be copied (e.g., using `strdup`).

Insertion and overwriting

Instead of providing variants of every function to insert or overwrite characters, this behaviour is controlled by a mode stored in the `strb_t` object. The initial mode is `strb_insert`. It can be read and written by the following functions:

- `strb_getmode`
- `strb_setmode`

Changing the mode can fail if the requested mode is not supported by the library, in which case the current mode is unchanged. Any value returned by `strb_getmode` is accepted by `strb_setmode`. If successful, `strb_setmode` returns zero, otherwise `EOF`.

Positioning

Instead of passing a position parameter to functions, an internal position indicator is stored in the `strb_t` object. The initial position is the end of the string. It is updated by operations on the string, but it can also be read and written by the following functions:

- `strb_tell`
- `strb_seek`

Repositioning can fail if the requested position is not supported by the library, in which case the current position is unchanged. Any value returned by `strb_tell` is accepted by `strb_seek`. If successful, `strb_seek` returns zero, otherwise `EOF`.

Passing `strb_seek` a position greater than the string length is allowed and does not change the length. If characters are later written beyond the end of the string, then the length will be updated. The initial value of any intervening characters will be 0.

Editing

The position indicator is relevant to use of the following functions:

- `strb_putc`
- `strb_unputc`
- `strb_puts`
- `strb_nputs`
- `strb_vputf`
- `strb_putf`
- `strb_write`
- `strb_delto`

Use of these functions abstracts decisions about whether characters put into a `strb_t` object by general-purpose string producers will be appended to, prepended to, inserted between, or overwrite other characters.

The `strb_putc` function copies one character into the buffer at the current position and increments the position indicator. If the mode is `strb_insert`, then characters at the current position are first moved upward to make space; otherwise, no characters are moved. Additional storage is allocated if necessary. If successful, `strb_putc` returns the character written, otherwise `EOF`.

The `n` parameter to `strb_nputc` indicates the number of times to copy the specified character into the buffer. It is equivalent calling `strb_putc` the specified number of times with the same value of `c`. If successful, `strb_nputc` returns the character written, otherwise `EOF`.

The `strb_unputc` function restores one character behind the current position and decrements the position indicator. If the mode is `strb_insert`, then any characters in the buffer at the former position are moved downward to the new position; otherwise, no characters are moved. This function may substitute a smaller buffer at the implementer's discretion.

A call to a function that deletes characters or sets the position/mode will discard any restorable characters in the `strb_t` object. Only one character is guaranteed to be restorable, regardless of mode. If `strb_unputc` is called too many times without an intervening write, then it may fail. If successful, `strb_unputc` returns the character removed, otherwise `EOF`.

The `strb_puts` function copies a null terminated string into the buffer associated with a `strb_t` object at the current position as if by calling `strb_putc` for each character except null, which is not copied. The `strb_nputs` variant limits the number of characters copied. If successful, these functions return zero, otherwise `EOF`. (This is stricter than `fputs`, which returns only ‘a nonnegative value’ if successful.)

The `strb_putf` and `strb_vputf` functions generate characters under control of a format string, which are copied into the buffer at the current position as if by calling `strb_putc` for each character. If successful, these functions return zero, otherwise `EOF`. (This differs from `fprintf`, which returns ‘the number of characters transmitted, or a negative value’.)

Wide characters (typically UTF-16) can be converted to multibyte characters and copied into the buffer by using the standard `l` length modifier:

```
wchar_t wstr[] = L"wide string";
strb_putf(string, "%ls", wstr);
```

The `strb_write` function allows direct insertion of strings into the buffer, particularly by third-party functions which cannot be modified to operate on a `strb_t`. It returns a pointer to the position where the first character should be written. Because it returns an unqualified pointer, this function also provides a mechanism for avoiding casts when calling functions that do not accept the address of a `const char`.

The `n` parameter to `strb_write` indicates the number of characters expected to be written into the buffer (not including any null terminator). If the mode is `strb_insert`, then `strb_write` will move characters at the current position upward to make space; otherwise, no characters are moved. Additional storage is allocated if necessary to allow `n+1` characters to be written. The initial value of any new characters is 0. The position indicator is advanced by `n` characters.

After copying up to `n+1` characters (including any null terminator) into the buffer at the address returned by `strb_write`, the user may call `strb_wrote` to restore the character that was at offset `n` from the current position before the call to `strb_write`. `strb_wrote` has no effect after an intervening call that puts, deletes, or restores characters, or which sets the position.

The `strb_delto` function deletes characters between the current position and a position specified by the caller. If the mode is `strb_insert`, then the character at the higher of the two positions is moved to the lower position, and any following characters are moved the same distance; otherwise, no characters are moved. Afterwards, the value of the position indicator is the lower of the two positions. This function may substitute a smaller buffer at the implementer's discretion.

Passing `strb_delto` a position greater than the string length is allowed: `SIZE_MAX` or `(size_t)-1` can be used as a shorthand to delete all characters between the current position and the end of the string.

The position indicator is interpreted as a byte rather than a character offset, therefore a call to any of the above functions has the potential to partially overwrite a multibyte character. Overwriting or deleting characters can have the same effect.

Replacement

The position indicator does not affect the following functions because they replace the whole string:

- `strb_cpy`
- `strb_ncpy`
- `strb_vprintf`
- `strb_printf`

The `strb_cpy` function replaces the string in a buffer by copying a null terminated string and sets the position indicator to the number of characters copied. The `strb_ncpy` variant limits the number of characters copied. If successful, these functions return zero, otherwise `EOF`.

The `strb_printf` and `strb_vprintf` functions replace the string in the buffer by generating characters under control of a format string. The position indicator is set to the number of characters generated. If successful, these functions return zero, otherwise `EOF`. (This differs from `sprintf`, which returns ‘the number of characters written in the array, not counting the terminating null character, or a negative value’.)

Errors

Additional storage for the underlying character array may be allocated automatically as the string grows, for example because of calls to `strb_puts`. When using a character array passed to `strb_use` or `strb_reuse`, or an internal buffer of fixed size, any attempt to allocate more storage fails.

If any attempt at storage allocation fails, the state of the buffer, mode, and position indicator are as if the failed operation had never been attempted. This allows use in interactive software running on machines with limited physical memory and no swap file.

An error indicator is stored to allow deferred error handling. Its value can be read at any time by calling `strb_error`. An error can only be cleared explicitly (by calling `strb_clearerr`).

The error indicator is set as a side-effect of any function that returns an error value, including because of attempts to reposition to an unsupported index or select an unsupported editing mode. Consequently, it is a reliable indication of whether the current string is valid.

Redundancy

Many functions are strictly redundant, but the alternatives are often cryptic:

- `strb_dup(sb, str)` is equivalent to `strb_aprintf(sb, "%s", str)` or `strb_cpy(strb_alloc(0), str)`.
- `strb_ndup(sb, str, n)` is equivalent to `strb_aprintf(sb, "%.*s", n, str)` or `strb_ncpy(strb_alloc(0), str, n)`.
- `strb_vaprintf(sb, format, args)` is equivalent to `strb_vputf(strb_alloc(0), format, args)`.
- `strb_puts(sb, str)` is equivalent to `strb_putf("%s", str)` or `strcpy(strb_write(sb, strlen(str)), str), strb_wrote(sb)`.
- `strb_nputs(sb, str, n)` is equivalent to `strb_putf("%.*s", n, str)` or `strncpy(strb_write(sb, strnlen_s(str, n)), str, strnlen_s(str, n))`.
- `strb_putc(sb, c)` is equivalent to `strb_putf(sb, "%c", c)` or `*strb_write(sb, 1) = c`.
- `strb_vputf(sb, format, args)` is equivalent to `vsprintf(strb_write(sb, vsnprintf(NULL, 0, format, args)), format, args), strb_wrote(sb)`.
- `strb_cpy(sb, str)` is equivalent to `strb_printf(sb, "%s", str)`.
- `strb_ncpy(sb, str, n)` is equivalent to `strb_printf(sb, "%.*s", n, str)`.
- `strb_nputc(sb, c, n)` is equivalent to `memset(strb_write(sb, n), c, n)`.

Some of the above equivalencies would have undefined behaviour if `strb_write` or `strb_alloc` returned null, therefore they aren't suitable for general use.

Implementation limits

When strings are simply character arrays, there is no limit on the number of characters in a string (except the natural limit of `SIZE_MAX-1`), nor on the maximum number of strings at run time.

A freestanding implementation of C is not required to provide the `malloc`, `strdup`, and `free` functions. Removing the requirement for dynamic storage allocation from string handling is desirable since it allows predictable runtime behaviour and performance guarantees.

To allow implementations of the proposed new string type that do not reallocate storage, it may be useful to expose the maximum number of characters that the library guarantees can be inserted into a string (analogous to `FILENAME_MAX`), and the maximum number of strings that can exist simultaneously (analogous to `FOPEN_MAX`). This allows implementations to statically allocate a finite number of objects of fixed size. Different limits might apply to objects managing external arrays.

These limits are not intended to encourage arbitrary restrictions on usage of strings but to allow implementers targeting resource-constrained platforms to provide the standard functions, albeit in a suitably restricted form.

There is a danger that macros analogous to `FILENAME_MAX` and `FOPEN_MAX` could be abused by programs, but on balance I consider it better to expose rather than hide implementation limits.

Example usage

List of fruit

The program to convert an array of integer indices into a comma-separated list of names has been reworked to use the proposed new standard functions:

```
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;
}
```

Buffer reuse for successive strings

The following code closely resembles part of a program written in 2001. Consequently, it is not an exemplar of good usage:

```
void append_to_csv(strb_t *const csv, char const *const value)
{
  if (strb_tell(csv) != 0)
    strb_putc(csv, ',');

  strb_puts(csv, value);
}

bool build_ships_stringset(strb_t *const output_string,
  char const *const graphics_set,
  bool const include_player, bool const include_fighters)
{
  /* Build string suitable to pass to stringset_set_available() */
  strb_delto(output_string, 0);

  _Optional strb_t *ship_name = strb_alloc(0);
  if (!ship_name) return false;

  if (include_player)
  {
    get_shipname_from_type(ship_name, graphics_set, ShipType_Player);
    strb_puts(output_string, strb_ptr(ship_name));
  }

  if (include_fighters)
  {
    for (ShipType i = ShipType_Fighter1; i <= ShipType_Fighter4; i++)
    {
      strb_delto(ship_name, 0);
      get_shipname_from_type(ship_name, graphics_set, i);
      append_to_csv(output_string, strb_ptr(ship_name));
    }
  }

  bool success = !strb_error(ship_name) && !strb_error(output_string);
  strb_free(ship_name);
  return success;
}
```

It would be more efficient to avoid allocating an intermediate buffer rather than reusing it:

```
bool build_ships_stringset(strb_t *const output_string,
  char const *const graphics_set,
  bool const include_player, bool const include_fighters)
{
  if (include_player)
    get_shipname_from_type(output_string, graphics_set, ShipType_Player);

  if (include_fighters)
  {
    for (ShipType i = ShipType_Fighter1; i <= ShipType_Fighter4; i++)
    {
      if (strb_tell(output_string) != 0)
        strb_putc(output_string, ',');

      get_shipname_from_type(output_string, graphics_set, i);
    }
  }
  return !strb_error(output_string);
}
```

Whether such optimisations can be made depends on whether it is feasible to modify third-party interfaces, and how much time is available for refactoring.

Direct append from an external source

```
if (messagetrans_lookup(&messages, token,
                        NULL, 0, &msgsize, 1, id_string) == NULL)
{
  _Optional char *const outtail = strb_write(output_string, msgsize - 1);

  if (outtail &&
      messagetrans_lookup(&messages, token,
         outtail, msgsize, NULL, 1, id_string) == NULL)
  {
    strb_wrote(output_string);
    return true;
  }
}
return strb_puts(output_string, token) >= 0;
```

Undo truncation for error handling

```
/* Try to recreate the top-level data structure again. */
old_dir_list = iterator->dir_list;
linkedlist_init(&iterator->dir_list);
strb_setpos(iterator->path_name, iterator->path_name_len);
strb_putc(iterator->path_name, '\0');

e = enter_dir(iterator);
if (e == NULL)
{
  /* Destroy the old data structures on success. */
  free_levels(&old_dir_list, NULL);
}
else
{
  /* Restore the previous state on error */
  iterator->dir_list = old_dir_list;
  strb_unputc(iterator->path_name);
}
```

Undo truncation to reinstate a leaf name

```
/* Remove the leaf name of the current directory from the path */
strb_setpos(iterator->path_name, ancestor->path_name_len);
strb_putc(iterator->path_name, '\0');

/* Try to refill the buffer with catalogue entries for the ancestor
   directory */
{
  DirIteratorLevel *tmp = ancestor;
  e = refill_buffer(iterator, &tmp);
  assert(tmp != NULL);
  ancestor = tmp;
}

/* Reinstate the leaf name of the current directory */
strb_unputc(iterator->path_name);
```

Undo an append to replace a leaf name

```
if (strb_putc(scan_data->save_path, '.') < 0)
{
  RPT_ERR("NoMem");
  return false;
}
scan_data->save_path_len = strb_len(scan_data->save_path);
```

And then later:

```
/* Remove the previous sub-path (does nothing if already undone) */
strb_delto(scan_data->save_path, scan_data->save_path_len);
e = append_to_string_buffer(scan_data->save_path,
                            scan_data->iterator,
                            diriterator_get_object_sub_path_name);
```

Strings from a format string

The standard function `snprintf` is commonly used to construct strings with automatic storage duration. Such code is prone to truncation, stack overflow, and programmer errors.

For example:

```
char kernel[KERNEL_SIZE];
int c = snprintf(kernel, KERNEL_SIZE, kernel_format, mem_size);
ASSERT((c > 0) && (c <= KERNEL_SIZE), "Source kernel truncated");
```

The above check for truncation is incorrect because the character count returned by `snprintf` does not include the null terminator. This error only manifests in an edge case and is therefore likely to go unnoticed.

The same code could be rewritten to use the proposed new functions like this:

```
char kernel[KERNEL_SIZE];
strb_t *s = strb_use(&(strbstate_t){}, sizeof kernel, kernel);
int err = strb_printf(s, kernel_format, mem_size);
ASSERT(!err, "Source kernel truncated");
```

Or, precluding future use of the `char` array via `strb_t`, like this:

```
char kernel[KERNEL_SIZE];
int err = strb_printf(strb_use(&(strbstate_t){}, sizeof kernel, kernel),
                      kernel_format,
                      mem_size);

ASSERT(!err, "Source kernel truncated");
```

Or, precluding future use of the `char` array except via `strb_t`, like this:

```
strb_t *kernel = strb_use(&(strbstate_t){},
                          KERNEL_SIZE,
                          (char [KERNEL_SIZE]){});

ASSERT(!strb_printf(kernel, kernel_format, mem_size),
       "Source kernel truncated");
```

The above examples reduce the likelihood of programmer error by simplifying checks for truncation.

The rewritten code also maintains the property that all objects have automatic storage duration. Subsequently appending to the generated string would use any remaining memory in the `char` array but would not allocate any extra storage. Such code would therefore be suitable for freestanding implementations.

The likelihood of stack overflow can be reduced by substituting objects with static storage duration (using syntax proposed by Romero in N2530 [34]):

```
strb_t *kernel = strb_use(&(static strbstate_t){},
                          KERNEL_SIZE,
                          (static char [KERNEL_SIZE]){});

ASSERT(!strb_printf(kernel, kernel_format, mem_size),
       "Source kernel truncated");
```

Often, it is impossible to predict the maximum size of a generated string. In such cases, simpler code is preferable:

```
_Optional strb_t *kernel = strb_aprintf(kernel_format, mem_size);
defer strb_free(kernel);
ASSERT(kernel, "No memory for source kernel");
```

The above example eliminates the possibility of truncation, but such code would only be suitable for hosted implementations.

Rationale

Encapsulation

Weak encapsulation of any interface is a genie that cannot be put back into the bottle. However, encapsulation hasn't historically been a big concern of C programmers compared to performance.

`strb_t` objects that wrap only a pointer are likely to have a size of 20 bytes in a 32-bit execution environment or 40 bytes on a 64-bit environment (like `mtx_t`) and could therefore practicably be assigned automatic storage duration. This looks tempting.

It could be argued that `mtx_t` provides a good precedent to follow in general, but under-specification of the semantics of copying objects of that type was raised by Sebor in N2191 [33]. Many of the issues raised in DR 493 would also apply to `strb_t` if it were a complete type.

Nevertheless, specifying `strb_t` as a complete type could have benefits:

- Simpler implementations.
- Unlimited number of `strb_t` objects without use of `malloc`.
- No need to call a destructor when managing a user-specified array or a fixed-size buffer.
- Control of storage lifetime and locality with respect to other objects.

These need to be weighed against:

- Making the requirement to call `strb_free` context-dependent could sow confusion.
- Locality of each `strb_t` to its associated buffer could be worse than if both had been allocated together.
- Initialisation functions would require an extra parameter to specify the target object.
- Under-specified or poorly understood rules for correct usage.
- Every `strb_t` object would be double-initialised or declared without an initialiser.

The requirement to call a destructor is made less onerous in many cases by the `defer` keyword proposed by N3199 [32]. To reap all the theoretical benefits of a complete type, programs would be restricted to very short buffers within each `strb_t` object or separately declared character arrays.

The following code is intended to illustrate some drawbacks of using a complete `strb_t` type:

```
foo_t *alloc_foo(const char *in, const char *out)
{
  foo_t *f = malloc(sizeof(*f));
  if (!f)
    return NULL;

  *f = (foo_t){.count = 14, .style = STYLE_ROMAN }; // first initialisations

  if (strb_init(&f->in, in) >= 0) // second initialisation of 'in'
  {
    if (strb_init(&f->out, out) >= 0) // second initialisation of 'out'
      return f;

    strb_destroy(&f->in);
  }

  free(f);
  return NULL;
}
```

Here is the same code using an incomplete `strb_t` type (as specified):

```
foo_t *alloc_foo(const char *in, const char *out)
{
  foo_t *f = malloc(sizeof(*f));
  if (!f)
    return NULL;

  *f = (foo_t){.in = strb_dup(in), .out = strb_dup(out),
               .count = 14, .style = STYLE_ROMAN };

  if (f->in && f->out)
    return f;

  strb_free(f->in);
  strb_free(f->out);
  free(f);
  return NULL;
}
```

Use of an incomplete type means an extra level of indirection is needed when compared to a plain character array. However, it provides strong encapsulation and avoids the need to specify copy or move semantics. It also allows future extensions such as reference counting or deferred deallocation.

The size of a pointer is well-defined for a given target architecture, and many users like knowing the stored size of common types. If `strdup` returns a pointer, then it seems reasonable for `strb_dup` to do likewise. The semantics of pointers are already well-defined and understood.

The performance impact of using an incomplete type could be mitigated by hiding a definition of the underlying structure as a different type and casting `strb_t` to that type for use by inline functions. However, that may be unnecessary, given advances in link-time optimisation.

`strb_t` objects needn't be allocated by `malloc`: a pool of statically allocated objects could be used instead. Storage locality and allocation costs could be optimized by storing short strings in a member of `strb_t` and switching to `malloc` for longer strings, if at all. That would remove a level of indirection for many common use cases.

These choices depend on the characteristics of the target machine, whether it implements `malloc`, and how efficient that implementation is. That is why they are best left to the implementer. The user can help by passing the likely maximum length of a string to `strb_alloc`. In many cases, the result should be a single allocation encapsulating both `strb_t` and buffer.

The complete type `strbstate_t` is a compromise that removes the requirement for freestanding implementations to use a statically allocated pool of `strb_t` objects. It also means that implementations that *do* use such a pool won't waste any character array that is internal to their statically allocated objects when managing an external array.

It only makes sense to require an object of type `strbstate_t` when managing an external array because it is more ergonomic and efficient to store information about an internally allocated buffer in neighbouring storage. This also allows programmers who want to avoid the potential pitfalls of explicitly managing storage to do so.

Terminology

The identifiers specified by my proposal should not be in existing use, since:

> *Function names that begin with `str`, `mem`, or `wcs` and a lowercase letter are potentially reserved identifiers and may be added to the declarations in the `<string.h>` header.*

(7.33.17 of ISO/IEC 9899:2023, Programming languages — C)

Aside from the fact that identifiers like `str_dup` are not reserved, it seems undesirable to invent new identifiers that cannot be distinguished phonetically from existing ones.

Standard functions whose names incorporate 'init', 'destroy', 'create' and 'delete' (e.g., `tss_create` or `mtx_init`) all operate on objects whose storage is managed by the caller. Consequently, the corresponding 'destroy' functions (e.g., `mtx_destroy`) do not accept null. It didn't seem appropriate to reuse those terms for functions that *can* return null, or which *do* accept null as an argument.

There is no precedent in the standard for 'new' although it is commonly used elsewhere.

'Open' didn't seem quite right either: it suggests correctly that null can be returned, but in functions like `fopen`, the creation of a `FILE` object is a side-effect of opening a file, whereas creation of a `strb_t` object is explicit. (Another point against 'open' and 'close' is the aberration that `fclose` does not accept null.)

Ultimately, I chose names that suggest `strb_t` creation and destruction resemble `malloc` and `free`. The Linux kernel uses this pattern, for example in the names of `alloc_workqueue_attrs` and `free_workqueue_attrs`. Existing functions that allocate strings have diverse names (e.g., the 'a' in `asprintf` means allocate) so I also followed that precedent for similar functions to create `strb_t` objects.

The pairing of those functions with `strb_free` seems apt:

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

Standard functions whose names incorporate 'cpy' (copy), 'dup' (duplicate) 'put' (at a current position) or 'cat′ (concatenate) establish a precedent for the meaning of those terms. Reusing them helps brevity and familiarity. Concatenation doesn't necessarily imply appending, but that is implicit in the parameter order. It could also be argued that that putting implies overwriting, but I don't think that nuance is often relevant.

The choice of `strb_nputs` instead of `strb_putsn` is somewhat arbitrary since there is precedent for both. The 'n' prefix seems to predominate (in `strncpy`, `strncat`, etc.) where a character limit concerns the source rather than the target of an operation (as in `snprintf`). No 'n' prefix is used for functions such as `memcmp` and `memcpy`, where it is not needed for disambiguation. The main issue with 'nput' is that it resembles 'unput' (the presumed analogue of `ungetc`).

There is ambiguity about the meaning of 'printf' because `sprintf` overwrites a character array from the start, whereas `fprintf` overwrites characters at a current file position (often perceived as appending). I resolved that by inventing the term 'putf'.

The term ‘erase’ is used by both C++ and GLib. I considered using ‘remove’ instead of ‘erase’ since there is precedent for it in the C standard but decided the risk of confusion between `strb_remove` and `strb_free` would be too great. (This is also true of ‘delete’, but I side-stepped it by defining the semantics as delete-to.) Note that ‘rem’ is short for ‘remainder’ rather than ‘remove’ in the name of functions like `remquo`.

`strb_ptr` could be named `strb_get`, since it has a similar purpose to `tss_get`. I decided against that to allow stream-like ‘get’ functions (e.g., `strb_getc`) to be added in future.

`strb_set` could be an alternative name for `strb_nputc`, since its interface resembles `memset`. This would eliminate potential confusion with `strb_unputc` but doesn’t suggest stream-like semantics.

Initialisation

`strb_alloc` is omitted from the core interface to allow freestanding implementations to restrict their support for the new interface to allocation of a fixed number of `strb_t` objects of small size.

I considered basing the design of `strb_alloc` on `setvbuf`. Specifically:

> *If `buf` is not a null pointer, the array it points to may be used instead of a buffer allocated by the `setvbuf` function and the argument `size` specifies the size of the array; otherwise, `size` may determine the size of a buffer allocated by the `setvbuf` function.*

(7.23.5.5 of ISO/IEC 9899:2023, Programming languages — C)

Other sources are more explicit that the `size` argument is merely a hint. It should therefore be acceptable to pass 0 and get a default buffer size.

This satisfied several requirements in one function:

- Creation of an internal buffer whose initial size is specified by the caller (like `g_string_sized_new`) to reduce the need for subsequent reallocations.
- Creation of a buffer from pre-allocated storage (like `fmemopen`) to allow safe management of any character array.
- Creation of an empty internal buffer (like `g_string_new(NULL)`) without passing the address of an empty string.

However, I decided against it:

- The interface is complex to document, implement and understand.
- It would be tempting to write `strb_alloc(0, 0)` instead of `strb_alloc(0, NULL)` or `strb_alloc(0, nullptr)`, which would be opaque.
- The name `strb_alloc` does not indicate possible use of an array.
- It creates a new class of run-time error if an implementation (conformant or not) requires an array to be passed.
- It’s easier to discuss inclusion of functions than nullability of parameters.
- Null pointers are commonly used as an error indicator; allowing them to be passed to `strb_alloc` removes the possibility of detecting mishandling of errors by static analysis.

Separating `strb_use` and `strb_alloc` clarifies programs and simplifies the common use case equivalent to `g_string_sized_new`.

The append functionality of `fmemopen` is useful for managing many persistent character arrays without requiring the same number of persistent `FILE` objects. However, requiring the array passed into `strb_use` to contain at least one null character would have drawbacks:

- It requires a linear search for the first null character (although this could end quickly).
- It creates a new class of run-time error (when no null character is present).
- Pre-initialising an array to an empty string is awkward or inefficient.

The last point might need further explanation. The following declaration does not initialise the first character of the array to null; instead, it initialises every character to null:

```
char buf[100] = "";
strb_t *sb = strb_use(sizeof buf, buf);
```

This is probably not what users expect or want – especially those who use automatic storage because it is "faster".

The following alternative only initialises the first character, but is awkward and error-prone:

```
char buf[100];
buf[0] = '\0';
strb_t *sb = strb_use(sizeof buf, buf);
```

It seems undesirable to force either alternative on every caller of `strb_use` that specifies an array. `strb_reuse` partially solves this by making it explicit when the content of pre-allocated storage is significant:

```
char buf[100] = "Hello";
strb_t *sb = strb_reuse(sizeof buf, buf);
if (sb) strb_puts(sb, " world");
```

It would be easy to allow `strb_dup(NULL)` with the same meaning as `g_string_new(NULL)`, but that would be redundant and irregular with respect to `strb_aprintf` and the existing `strdup` function.

Parameter order

The `strb_t` address is passed as the first parameter to `strb_nputs`, `strb_puts` and `strb_putc` instead of last. This mismatches the position of the `FILE *` parameter passed to functions `fputs` and `fputc`. However, the parameter order of `<stdio.h>` functions isn't consistent; regularity within the new interface seems more important.

It's tempting to think that a character count should always be passed before a related pointer so that the pointer can be declared as a variably modified type. However, I do not think that such declarations would be correct for my proposed interface.

For example, the following declaration specifies that `str` must point to an array of at least `n` characters:

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

This is not strictly correct because `strb_ncpy` does not read beyond the first null character it encounters. If the index of the first null character is less than `n-1`, then `str` need not point to an array of at least `n` characters.

Another reason not to adopt this style of declaration is that most standard functions instead pass parameters in the opposite order. However, I made an exception for `strb_use` to allow a diagnostic to be produced if the passed-in array is too short:

```
int main(void)
{
    char buf[1];
    strb_use(2, buf);
    return 0;
}
```

Output of GCC 13.2:

```
<source>: In function 'main':
<source>:12:5: warning: 'strb_use' accessing 2 bytes in a region of size 1 [-
Wstringop-overflow=]
   12 |     strb_use(2, buf);
      |     ^~~~~~~~~~~~~~~~
```

Positioning

`fgetpos` and `fsetpos` were added in ANSI C [26] to deal with files which are too large for `fseek` and `ftell` because of the limitation of encoding file positions in values of type `long`. This consideration doesn't apply to strings, whose length must always be representable by type `size_t`. Reuse of the terms 'seek' and 'tell' is intended to hint that unrestricted random access to strings is supported.

`strb_seek` is simpler than `fseek`, in that it cannot seek relative to a specified position. This avoids mixing signed and unsigned integer types. It can be combined with other functions to achieve the same effects as `fseek`:

- `strb_seek(sb, 0)` is equivalent to `fseek(stream, 0, SEEK_SET)`.
- `strb_seek(sb, strb_tell(sb) - 1)` is equivalent to `fseek(stream, -1, SEEK_CUR)`.
- `strb_seek(sb, strb_len(sb))` is equivalent to `fseek(stream, 0, SEEK_END)`.

A possible drawback is that any integer overflow (e.g., if `strb_tell(sb)` returns 0 in the example above) occurs in the calling code instead of in `fseek`. This would result in a huge position value, probably followed by storage allocation failure during the next 'put' operation.

I decided against specifying a complete opaque type to store string positions, along the lines of `fpos_t`. If provided, such a type seems likely to be abused (by treating it as a character index) by those who want direct control over positioning.

Even if a `wcsb_t` type and functions to read from a wide string buffer were added, a position type would still not be required. It is only necessary to incorporate `mbstate_t` in `fpos_t` because wide-oriented streams convert wide characters to multibyte characters and vice versa, as described in 7.23.2.6 of the C23 standard:

> *Each wide-oriented stream has an associated `mbstate_t` object that stores the current parse state of the stream. A successful call to `fgetpos` stores a representation of the value of this `mbstate_t` object as part of the value of the `fpos_t` object. A later successful call to `fsetpos` using the same stored `fpos_t` value restores the value of the associated `mbstate_t` object as well as the position within the controlled stream.*

The question of how unwritten characters exposed by seeking could be initialised is addressed by the Single UNIX Specification [27]:

> *The `fseek()` function allows the file-position indicator to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads of data in the gap will return bytes with the value 0 until data is actually written into the gap.*

This is a safe default value because null characters terminate strings. On the other hand, it does not affect the length reported by `strb_len` and is arguably less useful than clamping the position to the string length.

Repositioning can fail (like `fseek`) to avoid forcing implementations to use more bits to encode the internal position indicator than they use to encode the string length or buffer size. Thus, implementations need not allow arbitrary values of type `size_t` to be stored by `strb_seek` and retrieved by `strb_tell`.

This decision brings forward any error that would later occur if characters were subsequently written at an unsupported position, but it also prevents use of `SIZE_MAX` as a shorthand for 'delete from the end of the string'.

Deletion of characters

Data from GitHub indicates that `g_string_truncate` (12.2K files, 7th) is more widely used than `g_string_erase` (4.4K files, 11th), despite providing a subset of the latter's functionality. This reflects my experience with CBUtilLib, which only provides `stringbuffer_truncate`. Of the prior art examined, only GLib provides a function to delete an arbitrary range of characters. If no standard truncation function is provided, it must be easy to construct the equivalent operation; the same does not necessarily apply to deletion.

What the above figures do not show is how truncation is commonly used. I observe two idioms:

- `g_string_truncate(string, 0)` to empty the string.
- `g_string_truncate(string, string->len - n)` to remove `n` characters from the end of a string.

The first of these is equivalent to, but more efficient than, `g_string_assign(string, "");` the second suggests a gap between the intended and actual usage of GLib.

Users often want to undo one or more 'put' operations when the current position is still at the end of the written characters. This could be part of error handling, or a routine action such as appending the next file name within a directory to a path string.

Whether deleting characters is appropriate depends on the current mode: Undoing an overwrite (to the extent that's possible) only entails rewinding the position to the start of the overwritten characters; otherwise, following fields in a string that is intended to have a fixed layout would mistakenly be moved.

However, it's essential to minimise the need for users to write mode- or position-specific code, otherwise benefits to composability intended to emerge from use of the new interface may not be realised. That is why a delete function should not delete characters in `strb_overwrite` mode.

Deleting forward

A function to delete at the current position does not need to move the position indicator, but undo with a delete-forward function would need preceding calls to `strb_tell` and `strb_seek`:

```
strb_puts(sb, "foo");
if (strb_puts(sb, "bar")) {
   strb_seek(sb, strb_tell(sb) - strlen("foo"));
   strb_delfwd(sb, strlen("foo")); // no-op in overwrite mode
}
```

Often, the total number of characters put into the buffer is not readily available. An old position could be subtracted from the current position to obtain the number of characters to delete, but the resultant code begins to look tricky:

```
size_t start = strb_tell(sb);
if (fgets(foo, sizeof foo, f))
  strb_puts(sb, foo);

if (strb_puts(sb, "bar")) {
  size_t len = strb_tell(sb) - start;
  strb_seek(sb, start);
  strb_delfwd(sb, len); // no-op in overwrite mode
}
```

Deletion of matching substrings is another likely use case. Functions like `strstr` return a pointer to the start of a match, not the end, so deletion of a match feels natural with a delete-forward function:

```
_Optional const char *match = strstr(strb_ptr(sb), "foo");
if (match) {
  strb_seek(sb, match - strb_ptr(sb));
  strb_delfwd(sb, strlen("foo")); // no-op in overwrite mode
}
```

In favour of delete-forward:

- Closest resemblance to `g_string_erase` (and likely user expectations).
- Simple to specify that delete-forward is ignored in `strb_overwrite` mode.
- Deletion of matching substrings feels natural.

Against delete-forward:

- Undoing any previous 'puts' requires calls to `strb_tell` and `strb_seek`. Ideally, common operations should take place at the current position.

Deleting backward

A function to delete characters behind the current position must move the position indicator back by the same amount, otherwise successive calls would not delete contiguous substrings. Undo with a delete-back function would not need a preceding call to `strb_seek`:

```
strb_puts(sb, "foo");
if (strb_puts(sb, "bar"))
  strb_delbck(sb, strlen("foo")); // reposition-only in overwrite mode
```

The non-trivial version of the same code:

```
size_t start = strb_tell(sb);
if (fgets(foo, sizeof foo, f))
  strb_puts(sb, foo);

if (strb_puts(sb, "bar"))
  strb_delbck(sb, strb_tell(sb) - start); // reposition-only in overwrite mode
```

However, it feels less natural to seek the end of a match than the beginning:

```
_Optional const char *match = strstr(strb_ptr(sb), "foo");
if (match) {
  strb_seek(sb, match - strb_ptr(sb) + strlen("foo"));
  strb_delbck(sb, strlen("foo")); // reposition-only in overwrite mode
}
```

In favour of delete-back:

- Initial position is suitable for undo. No call to `strb_seek` is needed (unlike delete-forward).
- Repositioning is implicit (as for `strb_puts` et al.).
- Undoing a known number of 'puts' does not require a call to `strb_tell` (unlike delete-forward).

Against delete-back:

- Deletion of matching substrings feels unnatural.
- Less resemblance to `g_string_erase`.
- Complex to specify that delete-back only updates the position indicator in `strb_overwrite` mode.

Deleting to an absolute position

To avoid choosing between delete-back and delete-forward, a delete-to function could delete all characters between the current position and a specified absolute position:

```
strb_puts(sb, "foo");
if (strb_puts(sb, "bar"))
  strb_delto(sb, strb_tell(sb) – strlen("foo")); // reposition-only in overwrite
mode
```

The non-trivial version of the same code:

```
size_t start = strb_tell(sb);
if (fgets(foo, sizeof foo, f))
  strb_puts(sb, foo);

if (strb_puts(sb, "bar"))
  strb_delto(sb, start); // reposition-only in overwrite mode
```

The delete-match use case:

```
_Optional const char *match = strstr(strb_ptr(sb), "foo");
if (match) {
  strb_seek(sb, match - strb_ptr(sb));
  strb_delto(sb, strb_tell() + strlen("foo")); // no-op in overwrite mode
}
```

In favour of delete-to:

- Multi-purpose.
- Initial position is suitable for undo. No call to `strb_seek` is needed (unlike delete-forward).
- Repositioning is implicit for undo (as for `strb_puts` et al.).
- Undoing an unknown number of 'puts' requires just one call to `strb_tell` (fewer than delete-back).

Against delete-to:

- Undoing a known number of 'puts' requires a call to `strb_tell` (more than delete-back).
- Passing an absolute position to a delete function could cause confusion, depending on its name.
- Deleting a matching string requires a call to `strb_tell` or reuse of a stored/calculated current position.
- Complex to specify that delete-to only updates the position indicator in `strb_overwrite` mode.

Deleting to a signed offset

Another alternative would be to specify a delete-relative function that accepts a signed offset (like `fseek`) to delete relative to the current position:

```
strb_puts(sb, "foo");
if (strb_puts(sb, "bar"))
  strb_del(sb, -(int)strlen("foo")); // reposition-only in overwrite mode
```

The non-trivial version of the same code:

```
size_t start = strb_tell(sb);
if (fgets(foo, sizeof foo, f))
  strb_puts(sb, foo);

if (strb_puts(sb, "bar"))
  strb_del(sb, -(int)(strb_tell(sb) - start));
```

The delete-match use case:

```
_Optional const char *match = strstr(strb_ptr(sb), "foo");
if (match) {
  strb_seek(sb, match - strb_ptr(sb));
  strb_del(sb, (int)strlen("foo")); // no-op in overwrite mode
}
```

Casts in the above examples bring forward a conversion from `size_t` to `int` and suppress any diagnostic messages that might otherwise be issued. This reduces the likelihood of the result of the conversion being unrepresentable from a certainty to improbable.

For delete-relative:

- Multi-purpose.
- Initial position is suitable for undo. No call to `strb_seek` is needed (unlike delete-forward).
- Repositioning is implicit (as for `strb_puts` et al.).
- Undoing a known number of 'puts' does not require a call to `strb_tell` (unlike delete-forward).
- Deletion of matching substrings feels natural.

Against delete-relative:

- Often requires a conversion to a signed type that could produce an implementation-defined result or raise an implementation-defined signal.
- Limits the number of characters that can be deleted to half the theoretical maximum string length.
- Consistent use of `size_t` seems a desirable property to maintain.
- Complex to specify that delete-relative only updates the position indicator in `strb_overwrite` mode.

Conclusions

Different functions suit different use cases:

- Delete-forward or delete-relative suit deletion of matching substrings.
- Delete-backward or delete-relative suit undoing a known number of puts.
- Delete-to suits undoing an unknown number of puts.

I dismissed delete-relative because of its intractable signed/unsigned type issues.

If a `strb_erase` function is provided then it must have the same delete-forward behaviour as `g_string_erase` and `std::string:erase`, to avoid surprise. Minimally, the name of a delete-back or delete-to function needs to make clear that its semantics are nothing like `g_string_erase`. (Implementing the exact semantics of `g_string_erase` would raise the question of what effect it should have on the position indicator, if any.)

I decided to specify delete-to (as `strb_delto`) instead of delete-forward or delete-backward because it can do either. Deleting from an arbitrary position requires the caller to know that position. If the start position is known, then it should be trivial to calculate the end position (or vice versa).

Using delete-backward to undo 'put' operations requires the caller to calculate the number of characters to be removed. Except in trivial cases, it is less error-prone and more efficient to pass an old result of `strb_tell` to a delete-to function. Removing the need to maintain running totals is one of the goals of my proposal.

Delete-backward is still important, but better served under a different name. The standard function `ungetc` pushes one character back onto an input stream. This concept has been extended to output streams by providing a `strb_unputc` function, with the intention of replacing usage like `g_string_truncate(string, string->len - 1)`.

`strb_unputc` has another use: if `strb_putc` was used to truncate a string by writing a null character, then `strb_unputc` can be used to 'undo' truncation by reinstating the original character! The original character must have been stored in `strb_overwrite` mode. This completes the functionality of `stringbuffer_undo`.

The first character appended after `stringbuffer_truncate` overwrites the null terminator stored by that function, whereas a character appended after `strb_putc(sb, '\0')` does not. I don't foresee that being important in practice because such append operations also prevent truncation being undone by `stringbuffer_undo` (or `strb_unputc`).

Direct insertion

It must be straightforward and efficient to implement all the proposed write functions (`strb_puts`, `strb_putf`, etc.) using a combination of direct insertion and existing standard functions.

This serves several purposes:

- Proves the usability of direct insertion.
- Avoids divergence in time or space complexity between direct insertion and other algorithms.
- Reduces implementation and verification costs.
- Functions can be documented in terms of other functions.

A null character may appear anywhere in an array – not just at the end. For example, the following call to `sprintf` creates a string with a null character at both ends:

```
sprintf(buf, "%c.", 0);
```

A null character may also appear anywhere in the buffer of a `strb_t` object, since it is easy to insert one by calling `strb_putf(sb, "%c", 0)`. It would be hard to prevent such misuse. Consequently, `strb_len(sb)` and `strlen(strb_ptr(sb))` can return different results. However, this is rarely the desired outcome of insertion of a null-terminated string.

There are three ways of implementing insertion of a null-terminated string into an array:

1. Move the tail of the destination string by the length of the source string. Keep a copy of the destination character to be overwritten by the source string's terminator, then restore it after the insertion.
2. Move the tail of the destination string by one more than the length of the source string, then move the tail back by one character after the insertion.
3. Move the tail of the destination string by the length of the source string. Duplicate the source string, insert all its characters except the terminator, then destroy the duplicate.

Unfortunately, all those solutions require a three-step process, which introduces the potential error of calling a 'prepare' function but neglecting to call the corresponding 'finish' function.

3 is not 'direct' insertion, so it can be dismissed. 2 requires extra memory accesses which could be nearly as inefficient as copying from an intermediate buffer. That leaves only method 1.

Restoring an overwritten character could be done explicitly (by a 'finish' function) or implicitly (before the next read or write occurs). Implicit restoration could also be a side-effect of calling `strb_ptr`; any value it previously returned would be invalid after an insertion anyway because the buffer's address might have changed. However, it seems unintuitive for `strb_ptr` to modify the string.

A three-step process is also required if fewer characters are written into the buffer than expected (e.g., because `strb_write` was passed a maximum value like `MB_LEN_MAX`). In this scenario, excess characters must be deleted. Often, only one of those corrections needs to be applied (e.g., `c32rtomb` may output fewer than `MB_LEN_MAX` bytes but never appends a null terminator to its output). However, requiring user code to solve either problem would be more error-prone than requiring a call to a 'finish' function.

The interface provided by CBUtilLib requires adaptation:

- `stringbuffer_prepare_append` outputs the maximum number of characters that can be written. When only appending is allowed, it is trivial to return the amount of free space at the end of the buffer (which is often more than requested). When inserting, opening a bigger gap than requested would be counterproductive, so returning the available space is redundant.
- `stringbuffer_finish_append` receives the number of characters written and uses this to update the string length. When only appending is allowed, it doesn't matter if fewer characters were written than expected. When inserting a string of unknown length, it's impossible to restore the character overwritten by the terminator without keeping every potentially overwritten character (either by dynamically allocating another buffer, or by moving the tail of the destination string by double the length of the source string).

Keeping every potentially overwritten character until 'finish' is called could violate assumptions about the size of buffer required for string operations. This is particularly important when the buffer is an array specified by the user. Consequently, I removed the characters-written parameter from 'finish', and with it the feature of correcting for fewer characters having been written than expected.

Why not require the user to include space for a null terminator in the size passed to 'prepare', and then call `strb_unputc` instead of 'finish' to remove any unwanted terminator? Those aren't equivalent: in `strb_insert` mode, `strb_unputc` moves the tail of the string by one character, whereas `strb_wrote` never does that.

Unlike `stringbuffer_prepare_append`, the number of characters passed to `strb_write` need not include space for a terminator. This is safer, and better fits with the return value of functions like `snprintf`.

In conclusion, I considered this:

```
int strb_vputf(strb_t *sb, const char *format, va_list args)
{
  va_list args_copy;
  va_copy(args_copy, args);
  int len = snprintf(NULL, 0, format, args);
  if (len >= 0) {
    _Optional char *buf = strb_write(sb, len); // move tail by +len & keep buf[len]
    if (buf) {
      sprintf(buf, format, args_copy);
      strb_wrote(sb); // restore buf[len] overwritten by null
    }
  }
  va_end(args_copy);
  return len;
}
```

Preferable to this:

```
int strb_vputf(strb_t *sb, const char *format, va_list args)
{
  va_list args_copy;
  va_copy(args_copy, args);
  int len = snprintf(NULL, 0, format, args);
  if (len >= 0) {
    _Optional char *buf = strb_write(sb, len); // move tail by +len
    if (buf) {
      char tmp = buf[len]; // keep character expected to be overwritten
      sprintf(buf, format, args_copy);
      buf[len] = tmp; // restore character overwritten by null
    }
  }
  va_end(args_copy);
  return len;
}
```

Or this:

```
int strb_vputf(strb_t *sb, const char *format, va_list args)
{
  va_list args_copy;
  va_copy(args_copy, args);
  int len = snprintf(NULL, 0, format, args);
  if (len >= 0) {
    _Optional char *buf = strb_write(sb, len + 1); // move tail by +len + 1
    if (buf) {
      sprintf(buf, format, args_copy);
      strb_unputc(sb); // move tail by -1 and delete null character
    }
  }
  va_end(args_copy);
  return len;
}
```

Although all are valid usage of the interface as specified.

Error handling

The provision of a stored error indicator is based on several observations:

1. Writing comprehensive error-handling code is difficult.
2. The presence of such code (even if correct, and even if not executed) harms both programmer and execution efficiency.
3. The performance of code that fails typically doesn't matter.

The fact that the GPU driver, standard I/O streams, and other prior art such as OpenGL's `glGetError` function [28] provide this feature also indicate that it can be useful. In my experience, its usefulness increases with program size (since it is automatically preserved across function call and return).

The proposed interface only allows errors to be cleared explicitly (by calling `strb_clearerr`). This seems less likely to surprise users, but if a `strb_rewind` function is ever added then it must clear the error indicator to match `rewind`.

Choice of functions

I did not consider it necessary to specify 'nprintf' variants of the formatted string functions because the standard does not describe them for streams, and I didn't find any prior art other than `snprintf` and `vsnprintf`. I therefore assume that truncation is usually a 'bug' rather than a feature. This avoids likely confusion about whether the size parameter includes space for a null terminator.

Use of `strb_len` instead of `strb_tell` is likely to become an anti-pattern that harms composability, since they are equivalent when the position indicator is at the end (as it usually is). Nevertheless, I believe it's necessary to provide both functions.

The `strb_cpy` function fulfils two needs:

- An equivalent to `g_string_assign`.
- An equivalent to `cutils_astring_clear` (by replacing the current string with `""`).

Without `strb_cpy`, the current string could instead be replaced by:

```
strb_seek(sb, 0);
strb_delto(sb, SIZE_MAX); // delete to the end
strb_puts(sb, "empty");
```

But the above sequence might be considered too onerous.

Unfortunately:

- `strb_cpy` violates the general rule that all operations use the current position.
- It's not obvious what effect `strb_cpy` has on the current character position. (It moves to the end of the string.)
- There is a risk that producer functions accidentally call `strb_cpy` instead of `strb_puts`. This might not be noticed immediately, and it would limit the benefits to interoperability intended to result from use of the new interface.

I didn't want to specify an equivalent to `g_string_printf`, for the same reasons. According to my data, `g_string_printf` is used less often than `g_string_append_printf` (and probably in many cases where the latter could be substituted).

However, allocating the precious `strb_printf` name to mean something different from `g_string_printf` and orthogonal to `strb_cpy` seemed too risky. Instead, I chose to rename the functions that have stream-like semantics as `strb_putf` and `strb_vputf`, which is also more succinct.

`strb_nputc` is provided to satisfy a user request for a function to output a character a specified number of times [30]. It is also included to spark debate about the naming of `strb_unputc`.

Arguably, `strb_ndup`, `strb_aprintf` and `strb_vaprintf` need not be provided because GLib doesn't provide equivalents. However, `strb_ndup` seems like a useful equivalent to `strndup` that might reasonably be expected to exist. `strb_aprintf` and `strb_vaprintf` address a similar need to the GNU/BSD functions `asprintf` and `vasprintf`, but with the simpler interface favoured by the Linux kernel.

Not every GLib function has a direct equivalent. Most of the missing functionality can be replicated by combining calls to two or more functions belonging to the proposed interface. The number of calls required depends on context (e.g., successive insertions do not require multiple calls to `strb_seek`).

`g_string_new(NULL)`	`strb_alloc(0);`
`g_string_new(init)`	`strb_dup(init);`
`g_string_new_len(init, len)`	`strb_ndup(init, len);`
`g_string_sized_new(dfl_size)`	`strb_alloc(dfl_size);`
`g_string_free(string, TRUE)`	`strb_free(sb);`
`s = g_string_free(string, FALSE)`	`s = strdup(strb_ptr(sb));` `strb_free(sb);`
`g_string_free_and_steal(string)`	`s = strdup(strb_ptr(sb));` `strb_free(sb);`
`g_string_free_to_bytes(string)`	Not supported
`g_string_equal(v, v2)`	`strcmp(strb_ptr(v), strb_ptr(v2));`

`g_string_hash(str)`	`hash = 0;`
	`for (i = 0; i < strb_len(sb); ++i)`
	`  hash += strb_ptr(sb)[i];`

`g_string_assign(string, rval)`	`strb_cpy(sb, rval);`

`g_string_truncate(string, len)`	`strb_setmode(sb, strb_insert);`
	`strb_seek(sb, len);`
	`strb_delto(sb, SIZE_MAX);`

`g_string_insert_len(string, pos, val, len)`	`strb_setmode(sb, strb_insert);`
	`strb_seek(sb, pos);`
	`strb_nputs(sb, val, len);`

`g_string_append(string, val)`	`strb_seek(sb, strb_len(sb));` `strb_puts(sb, val) ;`
`g_string_append_len(string, val, len)`	`strb_seek(sb, strb_len(sb));` `strb_nputs(sb, val, len) ;`
`g_string_append_c(string, c)`	`strb_seek(sb, strb_len(sb));` `strb_putc(sb, c) ;`

`g_string_prepend(string, val)`	`strb_setmode(sb, strb_insert);`
	`strb_seek(sb, 0);`
	`strb_puts(sb, val);`

`g_string_prepend_c(string, c)`	`strb_setmode(sb, strb_insert);`
	`strb_seek(sb, 0);`
	`strb_putc(sb, c);`

`g_string_prepend_len(string, val, len)`	`strb_setmode(sb, strb_insert);`
	`strb_seek(sb, 0);`
	`strb_nputs(sb, val, len);`

<table>
<tr><td>g_string_insert(string, pos, val)</td><td>strb_setmode(sb, strb_insert);
strb_seek(sb, pos);
strb_puts(sb, val);</td></tr>
<tr><td>g_string_insert_c(string, pos, c)</td><td>strb_setmode(sb, strb_insert);
strb_seek(sb, pos);
strb_putc(sb, c);</td></tr>
<tr><td>g_string_overwrite(string, pos, val)</td><td>strb_setmode(sb, strb_overwrite);
strb_seek(sb, pos);
strb_puts(sb, val);</td></tr>
<tr><td>g_string_overwrite_len(string, pos, val, len)</td><td>strb_setmode(sb, strb_overwrite);
strb_seek(sb, pos);
strb_nputs(sb, val, len);</td></tr>
<tr><td>g_string_erase(string, pos, len)</td><td>strb_setmode(sb, strb_insert);
strb_seek(sb, pos);
strb_delto(sb, pos + len);</td></tr>
<tr><td>g_string_replace(string, find, replace, limit)</td><td>strb_setmode(sb, strb_insert);
for (i = 0; i < limit || !limit; ++i)
{
  match = strstr(strb_ptr(sb), find) –
               strb_ptr(sb);
  if (!match) break;
  size_t pos = match - strb_ptr(sb);
  strb_seek(sb, pos);
  strb_delto(sb, pos + strlen(find));
  strb_puts(sb, replace);
}</td></tr>
<tr><td>g_string_ascii_down(string)</td><td>s = strb_write(sb, 0)
for (i = 0; i < strb_len(sb); ++i)
  s[i] = tolower(s[i]);</td></tr>
<tr><td>g_string_ascii_up(string)</td><td>s = strb_write(sb, 0)
for (i = 0; i < strb_len(sb); ++i)
  s[i] = toupper(s[i]);</td></tr>
<tr><td>g_string_printf(string, format, ...)</td><td>strb_printf(sb, format, ...);</td></tr>
<tr><td>g_string_append_printf(string, format, ...)</td><td>strb_seek(sb, strb_len(sb));
strb_putf(sb, format, ...);</td></tr>
</table>

Several ImageMagick functions seem overly specialized and therefore are not supported directly:

<table>
<tr><td>StringInfo *head =
SplitStringInfo(string_info, offset)</td><td>_Optional strb_t *head =
strb_ndup(strb_ptr(sb), offset);
strb_seek(sb, 0);
strb_delto(sb, offset);</td></tr>
<tr><td>SetStringInfoLength(string_info, length)</td><td>strb_setmode(sb, strb_insert);
strb_seek(sb, strb_len(sb));
if (length < strb_len(sb))
  strb_delto(sb, length);
else if (strb_write(sb,
                    length - strb_len(sb)))
  strb_wrote(sb, length - strb_len(sb));</td></tr>
<tr><td>SetStringInfoDatum(string_info, source)</td><td>strb_setmode(sb, strb_overwrite);
strb_seek(sb, 0);
if (strb_write(sb, strb_len(sb))) {
  memcpy(strb_ptr(sb), source, strb_len(sb));
  strb_wrote(sb, strb_len(sb));
}</td></tr>
<tr><td>StringInfo *clone =
   CloneStringInfo(string_info)</td><td>_Optional strb_t *clone =
   strb_dup(strb_ptr(sb));</td></tr>
<tr><td>ConcatenateStringInfo(string_info, source)</td><td>strb_seek(sb, strb_len(sb));
strb_puts(sb, strb_ptr(source));</td></tr>
<tr><td>SetStringInfo(string_info, source)</td><td>strb_cpy(sb, strb_ptr(source));</td></tr>
</table>

Return values

Non-negative or zero value to indicate success

The standard function `fputs` only returns 'a nonnegative value' if successful or '`EOF` if a write error occurs'. Checking the sign of the result or checking for the exact value `EOF` adds complexity to every caller. Specifying the return value of `strb_puts` more strictly as zero or `EOF` (like the return value of `fclose`) simplifies the interface without invalidating existing practice, since `EOF` is negative and zero is not.

For example, the following error checks are all valid for `strb_puts` but the last would be invalid for `fputs` (because a non-zero value can be positive, implying success):

```
if (strb_puts(sb, "hello") < 0)
  break; // error

if (strb_puts(sb, "hello") == EOF)
  break; // error

if (strb_puts(sb, "hello"))
  break; // error
```

In contrast, the following error check is invalid for both `strb_puts` and `fputs`:

```
if (strb_puts(sb, "hello") == -2)
  break; // unreachable
```

Substitution of `bool` as the return type of `strb_puts` would be insufficient to ensure that error checks against out-of-bounds values (such as -2) are discovered at run time, because values of type `bool` are implicitly promoted to integer type in expression contexts.

However, Clang would issue a warning about such malformed comparisons:

```
<source>:869:32: warning: result of comparison of constant -2 with expression of
type 'bool' is always false [-Wtautological-constant-out-of-range-compare]
  869 |     if (strb_puts(sb, "hello") == -2)
      |         ~~~~~~~~~~~~~~~~~~~~~~ ^  ~~
```

GCC could be configured to do likewise:

```
<source>: In function 'main':
<source>:869:32: warning: comparison of constant '-2' with boolean expression is
always false [-Wbool-compare]
  869 |     if (strb_puts(sb, "hello") == -2)
      |                                ^~
```

This advantage must be weighed against precedent for standard functions to return `int` and the equally common practice of using 'false' to indicate failure (with the notable exception of checked integer arithmetic macros introduced by C23).

In-band error values

It's tempting to specify that `strb_putf` and `strb_vputf` should return the number of characters written, like `sprintf` and `vsprintf`. This could be useful for callers who wish to update a running total although removing the need for callers to maintain such totals is one of the goals of my proposal. A total could be computed more reliably from the return value of two or more calls to `strb_ftell`.

In-band error values returned by `sprintf` are liable to be misinterpreted. It is also problematic to allow up to `SIZE_MAX` characters to be written if the return type only allows up to `INT_MAX` writes to be reported. Changing the return type to `size_t` isn't a solution because putting 0 characters is not an error case. Instead of requiring the caller to pass an extra parameter, `strb_putf` and `strb_vputf` do not output the number of characters written.

Returning a pointer

Some might question why `strb_cpy` and `strb_ncpy` do not return a pointer to the destination array like the `strcpy` and `strncpy` functions which they resemble.

1. It seems preferable to have a consistent return type and interpretation of return values for the whole interface.
2. A pointer returned by `strb_cpy` or `strb_ncpy` could not be dereferenced without first checking for null. This would preclude idiomatic usage such as is possible when calling `strcpy` or `strncpy`.

For example, this is safe:

```
char array[10];
puts(strcpy(array, "Hello"));
```

Whereas this would be unsafe:

```
_Optional strb_t *array = strb_alloc(10);
if (array)
  puts(strb_cpy(array, "Hello"));
```

Under some circumstances, it might be useful for functions to return a pointer to their input `strb_t` object instead of `int` or a pointer to the managed string.

Consider this variation of one of the previous examples:

```
_Optional strb_t *sb = strb_cpy(strb_use(&(strbstate_t){},
                                         KERNEL_SIZE,
                                         (char [KERNEL_SIZE]){}),
                                "Hello");
```

The `strb_use` function cannot return a null pointer, so it is safe to pass its return value directly to `strb_cpy`. In this case, losing track of the address of the `strb_t` object if `strb_cpy` returns null does not matter because objects constructed by `strb_use` do not need to be explicitly freed.

However, I believe that the potential confusion created by returning a pointer from functions such as `strb_cpy` ("Can it create a new object? Does it free the old object? Must the return value be assigned?") outweighs any idiomatic conveniences that it might enable.

Possible future directions

The most obvious addition would be support for wide character strings:

```
typedef struct wcsb_t wcsb_t;

_Optional wcsb_t *wcsb_alloc(size_t size);
void wcsb_free(_Optional wcsb_t *sb /*in*/);
const wchar_t *wcsb_ptr(const wcsb_t *sb /*in*/);
size_t wcsb_len(const wcsb_t *sb /*in*/);
```

These were omitted for brevity and because they don't substantially affect the design.

Type-generic string functions could be declared in a new header (e.g., `<tgstring.h>`), to hide the proliferation of different character types from users:

```
#define strb_len(sb) _Generic((sb), \
      wcsb_t *: wcsb_len, \
      strb_t *: strb_len, \
      const wcsb_t *: wcsb_len, \
      const strb_t *: strb_len)(sb)
```

Such a header file need not be restricted to defining aliases for the new string functions.

Functions could be added to read from the current position and advance by the number of characters read:

```
_Optional char *strb_gets(strb_t *sb /*in,out*/,
                          size_t n /*characters+1*/,
                          char str[n] /*out*/);

int strb_getc(strb_t *sb /*in,out*/);
int strb_vgetf(strb_t *sb /*in,out*/,
               const char *format /*in*/,
               va_list args /*in*/);

int strb_getf(strb_t *sb /*in,out*/,
              const char *format /*in*/,
              ...);
```

The following functions could instead read and convert characters from the start of a string:

```
int strb_vscanf(strb_t *sb /*in,out*/,
                const char *format /*in*/,
                va_list args /*in*/);

int strb_scanf(strb_t *sb /*in,out*/,
               const char *format /*in*/,
               ...);
```

However, most prior art does not provide functions such as those above, so their inclusion is questionable.

Using `const char *` as the parameter type for functions such as `strb_puts` is good for interoperability but prevents the error state from being automatically propagated from source to destination, where the source is another `strb_t` object.

It could be useful to add variants of such functions which instead accept `const strb_t *` for convenience and to allow error propagation. However, it would be difficult to extend variadic functions such as `strb_printf` to support the same functionality.

Implementation

The purpose of this paper is not to promote a new library or any single implementation of the proposed functionality. The design of the interface, the ideas that it embodies, and the use-cases described herein are of greater importance. Nevertheless, a prototype implementation can be found at https://github.com/chrisbazley/strb_t/.

The prototype can be configured to allocate `strb_t` objects from the heap, from a static pool of configurable size, or use only externally allocated storage. It can be used to give an idea of the amount of code likely to be required for a full implementation, or for different subsets of the specified functionality. It can also be used for benchmarking.

Acknowledgements

The author would like to recognize the following people for their ideas, help, feedback, and encouragement: James Renwick, Linus Torvalds, Philip Withnall, Luke Culpepper, Rob Genders, Achal Pandey, Nikunj Patel, Matthew Clarkson, Dennis Tsiang, Raffaele Aquilone, Lukas Zapolskas, Martin Uecker, Aaron Ballman, Jens Gustedt, Roberto Bagnara, Federico Serafini, and Elizabeth Bazley.

Legal notices

GPU driver snippets

CBUtilLib

ImageMagick

Linux kernel

POSIX

GNU/BSD

GLib

References

[1] BBC Microcomputer Disc Filing System User Guide
http://chrisacorns.computinghistory.org.uk/docs/Acorn/Manuals/Acorn_DiscSystemUGI2.pdf

[2] The Advanced Disc Filing System User Guide
https://chrisacorns.computinghistory.org.uk/docs/Acorn/Manuals/Acorn_ADFSUG.pdf

[3] FileCore - Phase 1 Functional Specification
https://www.marutan.net/wikiref/Acorn%20Registered%20Developer%20REFERNC/RO4/API/HTML/FILECORE.HTM

[4] Carnegie Mellon University – SEI CERT C Coding Standard – STR03-C. Do not inadvertently truncate a string
https://wiki.sei.cmu.edu/confluence/display/c/STR03-C.+Do+not+inadvertently+truncate+a+string

[5] Programmer's Reference Manual – Format of a sprite
http://www.riscos.com/support/developers/prm/sprites.html#22649

[6] CBUtilLib
http://starfighter.acornarcade.com/mysite/programming.htm#cbutillib

[7] fmemopen - open a memory buffer stream
https://pubs.opengroup.org/onlinepubs/9699919799/functions/fmemopen.html

[8] open_memstream, open_wmemstream - open a dynamic memory buffer stream
https://pubs.opengroup.org/onlinepubs/9699919799/functions/open_memstream.html

[9] open_memstream.c (C) 2007 Eric Blake
https://github.com/artetxem/mitzuli/blob/master/libraries/tesseract-android-tools/src/main/native/com_googlecode_leptonica_android/stdio/open_memstream.c

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

[11] glibc – libc - Bug 1995 - fprintf() + fmemopen() error (?)
https://sourceware.org/bugzilla/show_bug.cgi?id=1995

[12] PRINTF(3) — FreeBSD Manual Pages
https://man.freebsd.org/cgi/man.cgi?query=asprintf

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

[14] Elixir Cross Referencer – Linux kernel source
https://elixir.bootlin.com/linux/latest/source/include/linux/sprintf.h

[15] [PATCH] Implement kasprintf
https://github.com/spotify/linux/commit/e905914f96e11862b130dd229f73045dad9a34e8

[16] GLib – Wikipedia
https://en.wikipedia.org/wiki/GLib

[17] GLib.String
https://docs.gtk.org/glib/struct.String.html

[18] Index of /sources/glib/1.1/
https://download.gnome.org/sources/glib/1.1/

[19] GLib handle out of memory
https://stackoverflow.com/questions/16974254/glib-handle-out-of-memory

[20] Carnegie Mellon University – SEI CERT C Coding Standard – ERR02-C. Avoid in-band error indicators
https://wiki.sei.cmu.edu/confluence/display/c/ERR02-C.+Avoid+in-band+error+indicators

[21] ImageMagick History
https://imagemagick.org/script/history.php

[22] MagickCore String Methods
https://github.com/ImageMagick/ImageMagick/blob/main/MagickCore/string.c

[23] Programmer's Reference Manuals – SWI Calls – MessageTrans_Lookup
https://www.riscosopen.org/wiki/documentation/show/MessageTrans_Lookup

[24] GitHub code search results
https://github.com/search?q=%2Fg_string_overwrite%5B%5E_%5D%2F+path%3A*.c+language%3AC&type=code&ref=advsearch

[25] ANSI C Rationale – The rewind function
https://www.lysator.liu.se/c/rat/d9.html#4-9-9-5

[26] ANSI C Rationale – The fseek function
https://www.lysator.liu.se/c/rat/d9.html#4-9-9-2

[27] The Single UNIX ® Specification, Version 2 – fseek
https://pubs.opengroup.org/onlinepubs/7908799/xsh/fseek.html

[28] glGetError – OpenGL 4 Reference Pages
https://registry.khronos.org/OpenGL-Refpages/gl4/html/glGetError.xhtml

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

[30] C – How to repeat a char using printf? – Stack Overflow
https://stackoverflow.com/questions/14678948/how-to-repeat-a-char-using-printf

[31] _Optional: a type qualifier to indicate pointer nullability
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3089.pdf

[32] Improved __attribute__((cleanup)) Through defer
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3199.htm

[33] Effects of copying a mtx_t object
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2191.htm

[34] Allow compound literals of static lifetime inside body functions
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2530.pdf