A String Type for C

Document: N3210
Date: 2024-01-14
Author: Martin Uecker

Introduction

C needs a string type. To get started and stimulate discussion, we formulate an initial proposal and discuss some general design questions.

Design

Here, we propose to add a minimalistic, length-prefixed, UTF-8 encoded, and zero-terminated string type **string_t** and some helper functions.

The main design goal is to provide safer basis for computation with UTF-encoded strings. At the same time, the string type should be low-level enough to be relatively performant, suitable to replace the most common open-coded string operations, and be usable for interoperability with existing C code, other languages, and systems. To achieve this, we propose to define the type as an incomplete (opaque) structure type, but suggest to fully specify the representation of this type according to:

```
struct string {
     size_t size;
     char8_t data[/* size */];
};
```

This design is a compromise between discouraging unsafe manipulation and keeping the door open for low-level interoperability. By making the type incomplete and providing functions that return the length of string excluding the terminating null character (**string_length**) and a function that provides a const qualified pointer to the data (**string_cstr**), as well as additional helper functions, direct manipulation of these strings can be avoided. Nevertheless, at least where read-only strings are sufficient, this still allows simple replacement of existing char pointers with **string_t** pointers in existing APIs by adding calls to **string_cstr** where the strings is handed over to existing code. Changes to the memory handling logic are then not required. Null termination and UTF8 encoding, as well as access to a predefined representation will lead to good interoperability with existing code, other languages, and tools. If necessary, a completed version of the type can be defined by the user, which allows seamless extensions. Specifying the representation ensures that the type becomes part of the ABI, which helps with interoperability.

Alternatives

Alternatively, we could use a fully defined struct type. The functions **string_length** and **string_cstr** may then not be required (one then needs to decide whether the size / length should then include the terminating null character or not - both choices could be confusing).

Going into a different direction, one could also leave the representation undefined. The advantage would be that the implementation has more freedom to transparently implement various optimizations. If the internal encoding should also be left to the implementation, the function **string_cstr** that now provides direct access to the underlying string needs to be replaced by a different interface. One could then also consider a requirement that the user has to use special memory allocation and deallocation functions (**string_free**) for these types, to give the implementation even more freedom. In this case, the type would act as an opaque memory buffer, which then could also be achieved by adding the corresponding functions for **FILE*** which may be an interesting alternative.

Future Extensions

An extension, which seems important, is to support the new type in I/O functions by adding suitable conversion specifiers (or length modifiers).

Here, only a minimal set of additional functions is defined. More high-level interfaces, e.g. for searching, splitting, etc. should be considered to encourage use of this string type instead of using error-prone hand-coded string processing. Also some functions that help with the Unicode aspects will likely be required. The focus should be on a small set of functions that nevertheless provides the most important functionality.

In this proposal, we also do not specify a string view type (e.g. **strview_t**) that can point at substrings inside other strings. Nevertheless, such a type is required for the efficient implementation of many string processing operations. Such a type could be added as a future extension:

```
typedef struct {
     size_t size;
     char8_t *data;
} strview_t;
```

For most of the functions included in this proposal, such as **string_length, string_cstr, string_concat, string_compare, string_append** etc., it would then make sense to have versions (or replacements) that take arguments of type **strview_t**. Future extensions to the core language (e.g. wide pointers) could also play a role here (but this would delay adoption).

Acknowledgment: Various members of WG14 for insightful discussions.

Proposed Wording

7.26 String handling <string.h>

7.26.1 String function conventions

1 The header <string.h> declares ~~one type~~ **two types**, ... ~~The type~~ **One of the types** is size_t ...

The type

```
string_t
```

is an incomplete structure type. An object of this type has the same representation as an object declared with the following type:

```
struct string_t {
     size_t size;
     char8_t data[/* size */];
};
```

The type stores a valid UTF-8 encoded character string in **data** using **length** characters in the encoding including the terminating null character. When passing a pointer to an object of type **string_t** to any function in this subsection, it is undefined behavior when the pointed-to object does not conform to these requirements.

The **string_from_utf8, string_from_utf16, and string_from_utf32** functions

Synopsis

```
#include <string.h>
string_t * string_from_utf8(const char8_t *s);
string_t * string_from_utf16(const char16_t *s);
string_t * string_from_utf32(const char32_t *s);
```

Description

The **string_from_utfX** functions allocate an object of type **string_t** as if by a call to **malloc** and initializes it with the UTF-X encoded string pointed to by **s**. X can be 8, 16, or 32. If the encoding of **s** is not a valid UTF-X encoding with a terminating null character, the behavior is undefined.

Returns

The **string_from_utfX** functions return the allocated **string_t** object. The returned pointer can be passed to **free**. If no space can be allocated the functions return a null pointer.

The **string_cstr** function

Synopsis

```
#include <string.h>
const char8_t * string_cstr(const string_t *s);
```

Description

The **string_cstr** function returns a pointer to the first character of the null-terminated string stored in the object pointed to by **s**. Modifying the returned string is undefined behavior.

Returns

The **string_cstr** returns a pointer to the first character of the string stored in the string object.

The **string_length** function

Synopsis

```
#include <string.h>
size_t string_length(const string_t *s);
```

Description

The **string_length** function returns the number of characters in the encoding of the string excluding the terminating null character.

Returns

The **string_length** functions returns the number of characters in string.

The **string_concat** function

Synopsis

```
#include <string.h>
string_t * string_concat(const string_t *a, const string_t *b);
```

Description

The **string_concat** function returns a pointer to a new object of type **string_t** that stores a string that is the concatenation of the strings stored in the object pointed-to by **a** and the string stored in the object pointed-to by **b**. The new object is allocated as if by **malloc**.

Returns

The **string_concat** function returns a pointer to the allocated object. The returned pointer can be passed to **free**. If no space can be allocated the **string_concat** function returns a null pointer.

The **string_compare** function

Synopsis

```
#include <string.h>
int string_compare(const string_t *a, const string_t *b);
```

Description

The **string_compare** function returns an integer -1, 0, 1 depending on whether the string pointed to by **a** comes earlier, is at the same position, or comes later in the lexicographic order.

Returns

The **string_compare** function returns an integer -1, 0, 1.

The **string_append** function

Synopsis

```
#include <string.h>
string_t * string_append(const string_t * restrict *a,
                         const string_t * restrict b);
```

Description

The **string_append** function returns a pointer to a new object of type **string_t** that stores a string that is the concatenation of the strings stored in the object pointed-to by ***a** and the string stored in the object pointed-to by **b**. The new object is allocated as if by **malloc**. If no space can be allocated a null pointer is returned and *a is not modified, otherwise the object pointed-to-by ***a** is freed and the pointer to the newly allocated object is stored in ***a.**

Returns

The **string_append** function returns a pointer to the allocated object. The returned pointer can be passed to **free**. If no space can be allocated the **string_append** function returns a null pointer.

The **string_printf** function

Synopsis

```
#include <string.h>
string_t * string_printf(const char * restrict fmt, ...);
```

Description

The **string_printf** function is equivalent to the **printf** function, except that the output is written to a a new object of type **string_t.** The new object is allocated as if by **malloc**.

Returns

The **string_printf** function returns the allocated object or a null pointer if an output or encoding error occurred or if no space could be allocated. The returned pointer can be passed to **free**.

The **string_vprintf** function

Synopsis

```
#include <string.h>
string_t * string_vprintf(const char * restrict fmt, va_list arg);
```

Description

The **string_vprintf** function is equivalent to the **vprintf** function, except that the output is written to a new object of type **string_t**. The new object is allocated as if by **malloc**.

Returns

The **string_vprintf** function returns the allocated object or a null pointer if an output or encoding error occurred or if no space could be allocated. The returned pointer can be passed to **free**.