Nxxxx: Variable length prefixed length strings

Document #:	Nxxxx
Date:	2025-06-24
Project:	Programming Language C
Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

Null terminated byte strings (NTBS) have been the traditional way of implementing text strings in C almost since its inception. NTBSs have the upside of minimum possible storage overhead, but they cost more processor time to use, are unnecessarily heavy on CPU memory caches, are a major source of bugs and security vulnerabilities, and given how RAM capacities have exponentially exploded in recent decades, it is probably time for C to standardise a length prefixed byte string.

WG14 has been working on this for many years now in one form or another. Most recently, it has been around [N3306] *`strb_t`: A standard string buffer type* however that is actually orthogonal to length prefixed arrays as it concerns dynamic memory allocated string buffers. Rather, the most recent paper I am aware of specifically dealing with length prefixed variably sized arrays is [N3210] *A string type for C* in 2024. That proposed:

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

... with the `char8_t` array being required to be zero terminated for backwards compatibility.

WG14 discussion for that paper felt that spending a `size_t` overhead per 'modern string' was too much, and it was wondered if a more compact length prefix could be designed. It was suggested we use the same technique as what UTF-8 uses to create a variable length prefix to minimise overhead. However others were concerned that this would confuse UTF-8 parsers, and that could open security concerns.

There **is** encoding space left unused by UTF-8 however. Would this be sufficient for our needs? Let's find out!

Contents

1 Quick recap of UTF-8 encoding

UTF-8 uses the top bits of each octet to describe a variable length encoding of a UTF codepoint. The rules are simple:

1. If the top bit is zero, the remaining seven bits are codepoints 0-127.
2. If the top three bits are `110`xxyyy, there will be a second continuation octet of the form `10` yyzzzz.
3. If the top four bits are `1110`wwww, there will be two continuation octets of the form `10`xxxxyy, `10`yyzzzz.
4. If the top five bits are `11110`uvv, there will be three continuation octets of the form `10`vvwwww, `10`xxxxyy, `10`yyzzzz.

As continuation bytes always have the top bits set of `10`, you can always find the beginning of the current UTF-8 sequence from any pointer or index into an array by scanning backwards by up to three bytes. This is known as *self-synchronisation* and it is a useful property.

If you examine the coding above, you will see that these octet values can never appear in a legal UTF-8 sequence:

There are two illegal values at:

- `0xc0 (11000000)`
- `0xc1 (11000001)`

There are three illegal values at:

- `0xf5 (11110101)`
- `0xf6 (11110110)`
- `0xf7 (11110111)`

And finally, there are eight illegal values at:

- `0xf8 (11111000)`
- `0xf9 (11111001)`
- `0xfa (11111010)`

- 0xfb (11111011)
- 0xfc (11111100)
- 0xfd (11111110)
- 0xfe (11111101)
- 0xff (11111111)

The proposal is that we build a variable length prefix encoding which is 'UTF-8 aware' in the sense that a correct UTF-8 parser will never parse our variable length prefix as a valid UTF-8 sequence.

2 The proposed variable length prefix encoding

It is proposed that the layout in memory for a variable length array of uint8_t would be:

```
struct varuint8_t
{
  uint8_t length[/* 1, 2, 4 or 8 */];
  uint8_t data[/* decoded length */];
};
```

length would always be one of one, two, four or eight as some processors have SIMD optimisations for UTF-8 processing which could be reused here[1].

data would be *optionally* zero terminated. This is because varuint8_t is a variable length array of uint8_t which can *also* be treated as a variable length array of char8_t, as we shall see later.

2.1 Array lengths zero to seven

A leading octet in the range 0xf8-0xff would indicate a very short array length 0-7 long, as there is **three** bits of storage in a single byte prefix.

Examples:

- 0xf8 is a zero length array, and is a total of one octet of storage.
- 0xf9,0x78 is a one item length array, and is a total of two octets of storage.

If used to represent a NTBS, the added overhead is a worst case of 100% and a best case of 12.5%. If the trailing null isn't needed, it has the same overhead as a NTBS.

2.2 Array lengths eight to seventy-one

A leading octet 0xf5 would indicate a short array length 8-71 long. There is one continuation octet of the form 10xxxxxx which means there is **six** bits of storage in a two byte prefix.

Example:

- 0xf5,0x80,0x4e,0x69,0x61,0x6c,0x6c,0x20,0x44,0x00 = Niall D0 is an eight item length array, and is a total of ten octets of storage.

[1] https://lemire.me/blog/2020/10/20/ridiculously-fast-unicode-utf-8-validation/

If used to represent a NTBS, the added overhead is a worst case of 25% and a best case of 2.8%. If the trailing null isn't needed, it has a worst case of 12.5%.

2.3 Array lengths seventy-two to 262k

A leading octet `0xf6` would indicate an array length 72-262215 long. There are three continuation octets of the form `10xxxxxx`,`10yyyyyy`,`10zzzzzz` which means there is **eighteen** bits of storage in a four byte prefix. As with UTF-8, continuation octets are big endian in orientation.

If used to represent a NTBS, the added overhead is a worst case of 5.6% and a best case of approximately zero.

2.4 Array lengths 262k - 4 trillion

A leading octet `0xf7` would indicate an array length 262216-4398046773319 long. There are seven continuation octets which means there is **forty-two** bits of storage in an eight byte prefix. As with UTF-8, continuation octets are big endian in orientation.

At this scale, the added overhead over a NTBS is always approximately zero.

2.5 The remaining prefixes

For now, the leading octets `0xc0` and `0xc1` would be reserved for future expansion in the standard.

This author's personal preference for string views/slices would be fat pointers, however if the committee felt mutable string buffers were better and we need to additionally encode a reservation or capacity, that's what these prefixes ought to be used for.

3 UTF-8 support

The proposal would be that standard library functions could safely return a `char8_t *` from a `varint8_t *` in various ways e.g.

```
// Returns the size of the array for access
size_t varuint8_length(const struct varuint8_t *arr);

// Returns the size of the array for memory copying
size_t varuint8_sizeof(const struct varuint8_t *arr);

// Return a pointer to the uint8_t at the beginning of the array
// Returns nullptr if array is zero length
uint8_t *varuint8_front(struct varuint8_t *arr);

// Return a pointer to the uint8_t at the end of the array
// Returns nullptr if array is zero length
uint8_t *varuint8_back(struct varuint8_t *arr);

// Return a pointer to the uint8_t at idx into the array
// Returns nullptr if idx is outside array
uint8_t *varuint8_index(struct varuint8_t *arr, size_t idx);

// Return a NTBS if array is null terminated AND contains
```

```c
// no intermediate null values, nullptr otherwise
char8_t *ntbs_from_varuint8(struct varuint8_t *arr);

// Return a pointer to the char8_t at or preceding arr[idx]
// Returns nullptr if idx is outside array or there is no
// valid UTF-8 codepoint at that index
char8_t *char8_from_varuint8_index(struct varuint8_t *arr, size_t idx);
```

4 Language support

It is not proposed yet, however if the language could directly support `struct varuint8_t` values e.g. so they could be copied around by value and `sizeof` and `arr[N]` worked out of the box, that would be great.

5 Runtime overhead

The storage space overheads have already been indicated, but what sort of runtime overhead might there be?

I implemented `varuint8_length()` to see how bad it might be:

```
varuint8_length:
        ldrb    w1, [x0]
        mov     x2, x0
        and     x0, x1, 7
        cmp     w1, 247
        bhi     .L1
        cmp     w1, 245
        beq     .L8
        cmp     w1, 246
        beq     .L9
        mov     x0, -1
        cmp     w1, 247
        beq     .L10
.L1:                                    # Array lengths 0-7
        ret
.L10:                                   # Array lengths 262k-4t
        ldr     x1, [x2]
        and     x0, x1, 1056964608
        ubfx    x2, x1, 16, 8
        ubfx    x3, x1, 56, 6
        ubfiz   x2, x2, 30, 6
        orr     x0, x0, x2
        and     x2, x1, 17732923532771328
        orr     x2, x3, x2, lsr 42
        ubfx    x3, x1, 32, 8
        add     x0, x0, x2
        and     x2, x1, 69269232549888
        ubfx    x1, x1, 8, 8
        ubfiz   x3, x3, 18, 6
        orr     x2, x3, x2, lsr 28
        ubfiz   x1, x1, 36, 6
```

```
        add     x1, x1, 262144
        add     x1, x1, 72
        add     x1, x2, x1
        add     x0, x0, x1
        ret
.L9:                                    # Array lengths 72-262k
        ldr     w1, [x2]
        and     x0, x1, 4128768
        ubfx    x2, x1, 24, 6
        ubfx    x1, x1, 8, 8
        orr     x0, x2, x0, lsr 10
        ubfiz   x1, x1, 12, 6
        add     x1, x1, 72
        add     x0, x0, x1
        ret
.L8:                                    # Array lengths 8-71
        ldrb    w0, [x2, 1]
        and     x0, x0, 63
        add     x0, x0, 8
        ret
```

Godbolt: https://godbolt.org/z/dEnfM6454

On a modern out of order superscaler CPU, I reckon length of array lengths 8-71 would take approximately three ticks; 72-262k would take approximately five ticks; 262k-4t approximately eight ticks.

This isn't the single tick overhead of a simple `size_t` length prefix, but it isn't too bad either. For some SIMD architectures, it should be possible to implement `varuint8_length()` branch free too.

6 References

[N3210] Uecker, Martin
A string type for C
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3210.pdf

[N3306] Bazley, Chris
strb_t: A standard string buffer type
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3306.pdf