WG21 PROPOSAL

Why Span Is Not Enough

Document Number:	D4036R0
Date:	2026-02-28
Audience:	LEWG
Reply-to:	Vinnie Falco vinnie.falco@gmail.com

Table of Contents

Abstract

C++ has bytes. A contiguous region of bytes needs a type. A sequence of such regions needs another. This paper examines the types that predictably come to mind, and their consequences.

Revision History

R0: Feb 2026

- Initial version.

Disclosure

The author maintains Boost.Beast[7], a published HTTP and WebSocket library built on Boost.Asio[1]'s buffer model, and develops Capy, Corosio, Http, Beast2, and Burl[10] - libraries that define or consume buffer abstractions. The author published P4003R0[8]. The author holds a neutral position on the Networking TS (changed from positive). This body of work creates a bias toward dedicated buffer types. Such types have costs: one more vocabulary type to learn, and interoperability friction with code that uses raw `span<byte>`.

Credit Where Due

`std::span` is a well-established vocabulary type. It turns a pointer and a size into a single thing. Perfectly. The vocabulary need is profound and this paper does not propose to diminish it.

The question is whether `span` is also the right vocabulary for I/O buffer descriptors.

1. `span<byte>`

Question. How do we represent a contiguous region of bytes?

Answer. `span<byte>` . A pointer and a size. That works.

However...

Platform I/O requires an array, not one region:

Platform	Descriptor	Used By
POSIX	`struct iovec`	`readv()` / `writev()`
POSIX	`struct msghdr`	`sendmsg()` / `recvmsg()`
Windows	`WSABUF`	`WSARecv()` / `WSASend()`
Windows	`FILE_SEGMENT_ELEMENT`	`ReadFileScatter()` / `WriteFileGather()`
Linux	`struct iovec`	`io_uring_prep_readv()` / `io_uring_prep_writev()`

`span<byte>` can describe one region. Wrap it in a one-element array and `readv()` accepts it. But I/O rarely involves a single contiguous region. A message has a header and a body. A protocol has framing and payload. Sending two regions with `write()` means two syscalls. Sending them with `writev()` means one - this is scatter/gather I/O. `span<byte>` is an insufficient type for representing an array of buffers.

2. `span<span<byte>>`

Question. How do we represent several such regions?

Answer. A span of spans. `span<span<byte>>` .

A single buffer is a view of someone's data. The bytes exist somewhere - in a `vector` , in a memory-mapped page, in a stack array. The buffer borrows them. Non-owning is natural. The data has a natural owner elsewhere.

A buffer sequence is different. Nobody "naturally" has an array of `span<byte>` objects lying around. The sequence is an assembled grouping - a data structure constructed to collect regions together. Making it non-owning means the grouping itself cannot be stored, returned, or passed across an asynchronous boundary.

```
class message {
    span<span<byte>> buffers_; // borrows... what?
};

span<span<byte>> prepare_message(span<byte> hdr, span<byte> body) {
    span<byte> bufs[] = { hdr, body };
    return { bufs }; // dangling
}

void start_send(socket& s, span<byte> hdr, span<byte> body) {
    span<byte> bufs[] = { hdr, body };
    s.async_send(span<span<byte>>(bufs), callback);
    // returns immediately; bufs destroyed; dangling
}
```

3. `range<span<byte>>`

Question. How do we own a collection of byte regions?

Answer. Use a range. `vector<span<byte>>` , `array<span<byte>, N>` , any range whose value type is `span<byte>` .

Ranges solve the ownership problem: a `vector` owns its elements.

Ranges create a byte consumption problem. Consider a JSON stream arriving in two chunks:

```
// chunk 1 [100 bytes]: {"name":"Alice","age":30}{"name":"B
// chunk 2 [100 bytes]: ob","age":25}...

range<span<byte>> input = { chunk1, chunk2 };

// parser finds first complete object: {"name":"Alice","age":30}
// that is 26 bytes - consume them
//
// views::drop(input, 1) drops all of chunk1 (100 bytes) - too much
// views::drop(input, 0) drops nothing - too little
// no standard range operation removes exactly 26 bytes
```

The parse boundary (26 bytes) does not align with the buffer boundary (100 bytes). Consuming 26 bytes means advancing chunk 1 by 26 bytes - 74 remain - without touching chunk 2. No range adaptor does this. `std::ranges` [5] operates on elements. Parsing operates on bytes, not elements.

Incremental parsers with this need - JSON, XML, CSV, protobuf - go unserved.

4. `byte`

Question. What if we add byte-level algorithms to a range of `span<byte>` ?

Answer. The range is fine for ownership and iteration. The element type is not.

`span<byte>` already serves too many needs: serialization, cryptography, hashing, memory-mapped regions. If buffer sequences also use `span<byte>` , the type system cannot distinguish a buffer from any other byte span. A concept, an overload, or a constraint that separates “buffer in a sequence” from “hash input” or “encryption key” is impossible to write.

Boost.Asio

A separate type enables run-time safety checks:

Capability	Asio `mutable_buffer` [1]	`span<byte>`
Implementation-defined members	Possible	Closed
Detect dangling after reallocation	Possible	No
Future diagnostic aids	Possible	No
Conditional debug callback	`BOOST_ASIO_ENABLE_BUFFER_DEBUGGING`	No

Each time `span<byte>` appears in a function signature, it loses the safety capability.

5. Six Ecosystems Already Arrived Here

Six I/O ecosystems, designed independently, all arrived at similar solutions:

Ecosystem	Buffer Type	Layout
POSIX	`iovec`	`void*` + `size_t`
Windows	`WSABUF`	`ULONG` + `char*`
Asio	`const_buffer` / `mutable_buffer`	`void const*` + `size_t` , with range concepts[1]
libuv	`uv_buf_t`	`char*` + `size_t` [2]

Ecosystem	Buffer Type	Layout
Go	`net.Buffers`	scatter/gather over `[][]byte` [3]
.NET	`ReadOnlySequence<T>`	linked list of discontiguous `Memory<T>` segments[4]

Everybody converged on custom types independently.

6. The Final Straw

The committee already endorsed this principle.

P0298R3[6] introduced `std::byte` because `unsigned char` performed triple duty. Neil MacIntosh wrote:

> *"these types perform a 'triple duty'. Not only are they used for byte addressing, but also as arithmetic types, and as character types. This multiplicity of roles opens the door for programmer error"*[6]

> *"The key motivation here is to make byte a distinct type - to improve program safety by leveraging the type system."*[6]

`unsigned char` had the right size and alignment. The committee added `std::byte` anyway - same size, same alignment, but no arithmetic, no implicit conversions. The generic type's operations did not match the domain. The committee restricted the interface.

`span<byte>` performs double duty - general-purpose byte view and I/O buffer descriptor. A bespoke type restricts the interface to `data()` and `size()` . Same principle, one level of abstraction higher.

The precise fit is bespoke.

Almost There

`std::byte` kept the shift operators despite the stated goal of removing arithmetic. The principle was right. The execution left a gap.

7. Finally Correct

New buffer types give us the principled option. Only what we need: `data()` and `size()` .

`void*` , Not `byte*`

`void*` is maximally accepting and minimally permissive. Any pointer converts to it implicitly. The user must perform an explicit cast to go back. The asymmetry is by design.

Risk	`void*`	`byte*`	Cost
Requires `reinterpret_cast`	No	Yes	Invites superfluous casts
Dereferenceable	No	Yes	Invites accidental access
Pointer arithmetic	No	Yes	Invites accidental arithmetic
Assignable to `span<byte>`	No	Yes	Invites full span API misuse
Promises byte-level meaning	No	Yes	Invites false type assertions
Contradicts type erasure	No	Yes	Invites type erasure violations
C++17 only	No	Yes	Disinvites C users

A Buffer Sequence Is Distinct

Buffer sequences are not served by existing concepts. They are a new concept.

What the Standard Needs

- A read-only byte region type (`void const*` + `size_t`)
- A writable byte region type (`void*` + `size_t`)
- Concepts for sequences of read-only and writable byte regions
- Algorithms: total byte count, byte-granular slicing, copy between buffer sequences

The types already exist:

```
class mutable_buffer {
    unsigned char* p_ = nullptr;
    std::size_t n_ = 0;
public:
    mutable_buffer() = default;
    mutable_buffer(mutable_buffer const&) = default;
    mutable_buffer& operator=(mutable_buffer const&) = default;
    constexpr mutable_buffer(void* data, std::size_t size) noexcept
        : p_(static_cast<unsigned char*>(data)), n_(size) { }
    constexpr void* data() const noexcept { return p_; }
    constexpr std::size_t size() const noexcept { return n_; }
};

class const_buffer {
    unsigned char const* p_ = nullptr;
    std::size_t n_ = 0;
public:
    const_buffer() = default;
    const_buffer(const_buffer const&) = default;
    const_buffer& operator=(const_buffer const& other) = default;
    constexpr const_buffer(void const* data, std::size_t size) noexcept
        : p_(static_cast<unsigned char const*>(data)), n_(size) { }
    constexpr const_buffer(mutable_buffer const& b) noexcept
        : p_(static_cast<unsigned char const*>(b.data())), n_(b.size()) { }
    constexpr void const* data() const noexcept { return p_; }
    constexpr std::size_t size() const noexcept { return n_; }
};
```

These are the Networking TS[9] types.

8. Side by Side

Task	`span<byte>`	`mutable_buffer`
Construct from vector	`span<byte>{reinterpret_cast<byte*>(v.data()), ...}`	`mutable_buffer{v.data(), v.size()}`
Consume N bytes	`buf = span<byte>{buf.data() + n, buf.size() - n}`	`buf += n`
Detect dangling	Requires ABI Break	***see-below***

Safety feature:

```cpp
class mutable_buffer {
    void* p_ = nullptr;
    size_t n_ = 0;
    void(*check_)() = nullptr;
public:
    void* data() const { if(check_) check_(); return p_; }
    size_t size() const noexcept { return n_; }
};
```

Smaller to write, safer to use, open to diagnostics.

9. But

But this is standardizing Asio’s types

Yes. They earn their keep.

But `vector<span<byte>>` is enough

Users opt out of types which do not let them opt out of allocations.

But `mdspan` is enough

Buffer sequences only need one dimension. `mdspan` [5] provides several.

But `span<void>` is enough

Even if `span<void>` were possible, what remains after removing the impossible is `data()` and `size()`. That is just a less-capable `mutable_buffer`.

But `span<byte>` is enough

`span<byte>` is also a less-capable `mutable_buffer`. It is `span<void>` with added harm.

Suggested Straw Poll

> *LEWG agrees that a contiguous byte region descriptor for I/O should be a dedicated type, not* `span<byte>` .

Acknowledgements

The buffer model described here draws on twenty years of Asio's buffer sequence abstractions, due to Chris Kohlhoff.

References

1. Boost.Asio - Buffer types and buffer sequence requirements (Chris Kohlhoff). https://www.boost.org/doc/libs/release/doc/html/boost_asio.html
2. libuv - `uv_buf_t` buffer type. https://docs.libuv.org/en/v1.x/
3. Go standard library - `net.Buffers` (https://pkg.go.dev/net#Buffers). https://pkg.go.dev/
4. .NET System.IO.Pipelines - `ReadOnlySequence<T>` . https://learn.microsoft.com/en-us/dotnet/api/system.io.pipelines
5. C++ Working Draft - `span` (span.overview), `mdspan` (mdspan.overview), ranges (ranges). https://eel.is/c++draft/
6. P0298R3 - A byte type definition (Neil MacIntosh). https://wg21.link/p0298r3
7. Boost.Beast - HTTP and WebSocket built on Boost.Asio (Vinnie Falco). https://github.com/boostorg/beast
8. P4003R0 - (Vinnie Falco). https://wg21.link/p4003r0
9. N4771 - Working Draft, C++ Extensions for Networking. https://wg21.link/n4771
10. C++ Alliance - Capy, Corosio, Http, Beast2, Burl (Vinnie Falco). https://github.com/cppalliance