# The Need for Escape Hatches

| | |
|---|---|
| Document Number: | P4035R1 |
| Date: | 2026-05-01 |
| Intent: | Inform |
| Audience: | LEWG |
| Reply-to: | Vinnie Falco vinnie.falco@gmail.com<br>C++ Alliance Proposal Team |

Table of Contents



## Abstract


C++ should make the safe thing easy, and the unsafe thing possible.


---

## Revision History

### R1: May 2026 (pre-Brno mailing)

- Formatting corrections.

### R0: April 2026 (post-Croydon mailing)

- Initial version.

---

## 1. Disclosure

The author provides information and serves at the pleasure of the committee.

The author developed and maintains Capy [1] and Corosio and believes coroutine-native I/O is a practical foundation for networking in C++.

The author has published Boost libraries and has a stake in the project's success.

This paper asks for nothing.

---

## 2. The Need for Escape Hatches

Safe interfaces should be the default. They should validate input, maintain invariants, and protect users from misuse. However, C++ also needs an explicit path for trusted data - when the precondition is already satisfied at a boundary and re-validation is pure overhead.

The two paths differ in their contracts:

| | Safe path | Escape hatch |
|---|---|---|
| **Contract** | Wide | Narrow |
| **Naming** | Default | Explicit, marked |

Functions on the safe path should have wide contracts. Functions on the escape hatch carry narrow contracts.

The naming convention appears in the standard library, and the full pattern - naming and contract - appears in production Boost libraries, in the current `cstring_view` proposal, and in coroutine-based concurrency libraries. Four independent examples follow. A fifth section applies the pattern to `cstring_view` constructor design. A sixth examines what happens when the pattern is applied to the implicit `char const*` constructor.

---

## 3. Standard Precedent

The standard library provides `std::condition_variable` [2], which requires `std::unique_lock<std::mutex>`, and `std::condition_variable_any` [2], which works with any lockable type:

```
std::mutex mtx;
std::shared_mutex smtx;

// Constrained: only unique_lock<mutex>.
std::condition_variable cv;
std::unique_lock<std::mutex> lk(mtx);
cv.wait(lk);

// Broader: works with any lockable.
std::condition_variable_any cv_any;
std::shared_lock<std::shared_mutex> slk(smtx);
cv_any.wait(slk);
```

`condition_variable_any` is the broader, more general facility - the safe path that works with any lock type. `condition_variable` is the optimized path - a thin wrapper over `pthread_cond_t` that works with only one lock type. Under the escape-hatch principle, the broad facility should carry the default name and the optimized facility should carry the marked name.

The naming went the other way. N2406 [3] ("Mutex, Lock, Condition Variable Rationale") documents the design: `condition_variable` was intended as "as thin a wrapper as possible around that OS functionality," and `condition_variable_any` was the generalization built on top. The primary name followed the POSIX lineage, not the safety principle. The optimized primitive got the short name; the broader facility got the suffix.

The standard library already has the naming convention. It got it backwards because the principle was not articulated at the time.

---

## 4. Established Practice

Boost.URL [4] provides `pct_string_view`, a non-owning reference to a valid percent-encoded string. Construction from untrusted input validates the encoding and throws on failure:

```
// Safe default: validates percent-encoding.
pct_string_view s("Program%20Files");    // OK
pct_string_view bad("100%pure");         // throws
```

Internally, the library's own parser has already validated the encoding before extracting URL components. Repeating that validation would be redundant. A separate function bypasses it at the trusted boundary:

```
// Escape hatch: precondition already satisfied by the parser.
pct_string_view s = make_pct_string_view_unsafe(data, size, decoded_size);
```

Boost.URL has shipped this pattern with years of field experience: a validating default (`pct_string_view` [5]) and an escape hatch (`make_pct_string_view_unsafe` [5]) for trusted boundaries. Boost.Process (`basic_cstring_ref` [6]) and Boost.SQLite (`cstring_ref` [7]) independently implemented null-terminated string reference types, confirming the demand for the type that `cstring_view` proposes to standardize.

---

## 5. Application Level

On BSD-derived systems, directory iteration exposes a filename pointer and a filename length via `dirent` (`d_name` and `d_namlen`) FreeBSD `readdir(3)` [8] and FreeBSD `dirent.h` [9]. POSIX requires that path components are null-terminated and contain no embedded null bytes POSIX Base Definitions [10]. Rescanning each name in a validating constructor repeats work the operating system already did.

```
void visit_directory(DIR* dir)
{
    while (dirent* de = ::readdir(dir))
    {
        const char* p = de->d_name;
        std::size_t n = de->d_namlen;

        // Trusted boundary: OS already satisfied the precondition.
        cstring_view name = cstring_view::unsafe(p, n);
        consume(name);
    }
}
```

The safe path remains the default for untrusted input. The unsafe path exists for proven preconditions and zero additional runtime cost.

---

## 6. Structured Concurrency

Structured concurrency enforces the same discipline for asynchronous lifetimes that structured control flow enforces for execution order: activations nest, scopes nest, a parent outlives its children, and RAII works. Capy [1] provides `run` as the structured default and `run_async` as the explicit escape hatch:

```
recycling_frame_allocator alloc;

// Safe default: structured, caller suspends until child completes.
co_await run(ex, alloc)(my_coro());

// Escape hatch: fire-and-forget, caller continues immediately.
run_async(ex, alloc,
    []() { /* Result handler. */ },
    [](std::exception_ptr) { /* Error handler. */ }
)(my_coro());
```

The structured path is the default. The unstructured path requires a longer name and explicit handlers - the caller must decide up front how results and errors are delivered. The escape hatch still provides guardrails: a work guard keeps the execution context alive, a stop token enables cooperative cancellation, and typed handlers prevent silent result loss.

The consequences of misuse are higher in concurrency than in data validation - a wrong `pct_string_view` produces garbage data, while a wrong detached task produces data races. That severity is precisely why the escape hatch must be designed rather than left to raw `std::thread` or `std::async`, which offer none of these guardrails. Eliminating the escape hatch does not eliminate unstructured concurrency; it pushes programmers toward worse tools.

---

## 7. `cstring_view` Constructors

P3655R3 [11] ("cstring_view") proposes `std::cstring_view`, a non-owning view guaranteed to be null-terminated. The type fills a real gap: `std::string` owns and null-terminates, `std::string_view` does not own and does not null-terminate, and `cstring_view` does not own but does null-terminate. Over 2,100 independent implementations on GitHub confirm the demand. The `substr` split - one-argument returning `cstring_view`, two-argument returning `string_view` - and the deletion of `remove_suffix` show careful attention to the null-termination invariant.

P3566R2 [12] ("You shall not pass `char*`") independently arrives at the same pattern for `string` and `string_view`: deprecate the `char const*` constructor as an unbounded-range operation, add a bounded `char[N]` constructor for arrays, and provide an explicitly tagged `unsafe_length_t` replacement for the deprecated path. The escape-hatch structure - safe default, explicit opt-in - is identical to the pattern documented in Sections 2-5.

The escape-hatch pattern from Sections 2-5 applies directly to the constructor set. P3655R3's pointer-and-length constructor has a narrow contract:

```
constexpr basic_cstring_view(const charT* str,
    size_type len);   // Precondition: str[len] == '\0'.
```

Violating the precondition is undefined behavior. The design purchases O(1) construction for callers who already hold null-terminated data. Completely reasonable when the data is trusted.

LEWG designs are evaluated in a climate where safety is a first-order concern. A constructor whose public interface admits undefined behavior when a caller provides unterminated data will face scrutiny that a constructor with well-defined behavior for all inputs does not. This is the observable condition under which the design operates.

Two callers illustrate the tension:

```
void caller_a(DIR* dir)
{
    dirent* de = ::readdir(dir);
    const char* p = de->d_name;
    std::size_t n = de->d_namlen;

    // OS guarantees null-termination. O(1) desired.
    cstring_view name(p, n);
    consume(name);
}

void caller_b(std::span<const char> buffer)
{
    const char* p = buffer.data();
    std::size_t n = buffer.size();

    // Data origin unknown. Null-termination not guaranteed.
    cstring_view name(p, n);  // UB if not null-terminated.
    consume(name);
}
```

Caller A holds trusted data and wants zero-cost construction. Caller B holds untrusted data and needs validation. One constructor cannot serve both safely. The escape-hatch pattern resolves this:

```
// Wide contract: validates that str[len] == '\0'.
// Throws if not. No undefined behavior.
constexpr cstring_view(const charT* str,
    size_type len);

// Narrow contract: O(1), no scan.
// Precondition: str[len] == '\0'. UB if not.
static constexpr cstring_view unsafe(
    const charT* str, size_type len) noexcept;
```

The safe constructor scans. The cost is O(n). The `unsafe` factory trusts the caller and pays nothing. This is the same pattern as `pct_string_view` and `make_pct_string_view_unsafe` from Section 4 - safe by default, explicit opt-in at trusted boundaries.

C++ earns its reputation as a zero-cost abstraction language by letting programmers pay only for what they use. A type that validates on every construction - with no way to bypass validation at a trusted boundary - imposes cost the programmer cannot eliminate. The `unsafe` factory preserves this principle.

The choice of the name `unsafe` is deliberate. It mirrors the Rust keyword, where `unsafe` marks the boundary at which the programmer takes responsibility for invariants the language normally enforces. `cstring_view::unsafe` marks the same kind of boundary: the caller asserts null-termination that the constructor normally verifies. The cross-language resonance makes the API legible to developers familiar with either language.

P3655R3 also provides a templated constructor from an iterator pair:

```
template<class It, class End>
constexpr basic_cstring_view(It begin, End end);
```

The contract requires `*(begin + (end - begin)) == '\0'` - the iterator at position `end` must be dereferenceable. This breaks the standard C++ convention that `end` is past-the-end and not dereferenceable. Under a wide-contract model, three options present themselves:

- **A.** Require `[begin, end]` readable (inclusive), scan for the first null. Same model as the pointer-and-length constructor.
- **B.** Require only `[begin, end)` readable, scan the half-open range for the first null.
- **C.** Remove the constructor. No demonstrated use case requires it that the pointer-and-length constructor does not already serve. Real-world `cstring_view` usage originates from `const char*` (C APIs, OS calls, string literals) or from `std::string` (implicit conversion). Ship the type without this constructor; add it later when evidence of need emerges.

---

## 8. The Implicit Constructor

P3566R2's `char[N]` constructor is a genuine improvement. It captures bounds at compile time, eliminates `strlen` for string literals, and provides bounded safety for array sources.

The `char const*` constructor that P3566R2 deprecates is implicit. That property is load-bearing. Every function that accepts `string_view` or `cstring_view` today silently accepts `char const*` at the call site - no tag, no cast, no ceremony. Deprecating the implicit constructor does not degrade one API. It degrades every call site where a `char const*` crosses into a function taking a view type.

The standard library's own error-reporting interface returns `char const*`:

```
void log_error(std::string_view msg);

try { /* ... */ }
catch (std::exception const& e) {
    log_error(e.what());              // today
    log_error(std::string_view(       // under P3566
        unsafe_length, e.what()));
}
```

C library functions exhibit the same pattern. `std::getenv()`, `std::strerror()`, and `std::setlocale()` return `char const*` and always will:

```
void configure(cstring_view path);

configure(std::getenv("HOME"));       // today
configure(cstring_view(               // under P3566
    unsafe_length,
    std::getenv("HOME")));
```

P3566R2 documents that certain language constructs force array-to-pointer decay - notably, the ternary operator and initializer lists [12]. Both operands in `cond ? "first" : "second"` are string literals. Both are arrays. The language decays them to `char*` before overload resolution. The `char[N]` constructor cannot intercept:

```
void f(std::string_view s);

f(cond ? "first" : "second");         // today
f(cond ? "first"sv : "second"sv);     // under P3566
```

P3566R2's own migration data quantifies the scope. The Qt experiment reports that "all of Qt's runtime reflection APIs yield `const char*`, causing many warnings that one is unable to quickly address" [12]. The NVIDIA Omniverse migration required codebase-wide search and replace of `const char* x = "..."` declarations to `constexpr char x[] = "..."` [12].

The `char[N]` constructor adds a bounded path for literals. The `char const*` constructor is the only implicit path for runtime string sources - `std::exception::what()`, `std::getenv()`, `std::strerror()`, stored `c_str()` results, ternary expressions. Adding a path is not the same as closing one.

LEWG reviewed this evidence at Sofia (June 2025) and reached consensus against pursuing the full direction of P3566R1 for the standard library (SF/F/N/A/SA: 0/7/3/9/6) [13]. The committee did encourage further work in a profile context and on changing undefined behavior for NULL to erroneous behavior - directions that do not require closing the implicit constructor. The migration costs documented above explain the first outcome.

At Croydon (March 2026), LEWG polled P3655R3's constructor design directly [14]:

**POLL:** We would like to see array constructors in this paper for `cstring_view` before forwarding.

| SF | F | N | A | SA |
|---|---|---|---|---|
| 2 | 3 | 7 | 8 | 5 |

Weak consensus against.

**POLL:** We would like to remove the ctor that takes `(char*)` (only) from the proposal for now. (Due to safety concerns.)

| SF | F | N | A | SA |
|---|---|---|---|---|
| 1 | 5 | 1 | 8 | 9 |

Consensus against.

The committee declined both to require bounded array constructors before forwarding and to remove the `char const*` constructor on safety grounds.

---

## 9. Conclusion

The standard library already provides constrained defaults with explicit broader counterparts. Production libraries converge on the same pattern for trusted boundaries. The same principle governs concurrency structure. The evidence documents a recurring design value: safe by default, with explicit escape hatches where zero-cost composition requires them.

How explicit escape hatches interact with Hardening, Contracts, and Erroneous Behavior is a related question that deserves separate treatment.

---

## Acknowledgements

The author thanks Howard Hinnant for the `condition_variable` example and for the historical context in N2406. The author thanks Jonathan Wakely, Jan Schultke, Pablo Halpern, and Nevin Liber for discussion and feedback on safe and unsafe construction paths. The author thanks Peter Bindels, Hana Dusikova, Jeremy Rifkin, Marco Foco, and Alexey Shevlyakov for their work on `cstring_view`. The author thanks Giuseppe D'Angelo and Joshua Kriegshauser for the migration data cited in Section 7.

---

## References

[1] Capy

[2] C++ Working Draft, `condition_variable` https://eel.is/c++draft/thread.condition.condvar, `condition_variable_any` https://eel.is/c++draft/thread.condition.condvarany

[3] N2406, "Mutex, Lock, Condition Variable Rationale," Howard E. Hinnant

[4] Boost.URL

[5] Boost.URL, `pct_string_view.hpp`

[6] Boost.Process, `cstring_ref.hpp`

[7] Boost.SQLite, `cstring_ref.hpp`

[8] FreeBSD `readdir(3)`

[9] FreeBSD source, `sys/sys/dirent.h`

[10] The Open Group Base Specifications Issue 8

[11] P3655R3, "cstring_view," Peter Bindels, Hana Dusikova, Jeremy Rifkin, Marco Foco, Alexey Shevlyakov

[12] P3566R2, "You shall not pass `char*` - Safety concerns working with unbounded null-terminated strings" Marco Foco, Joshua Kriegshauser, Alexey Shevlyakov, Giuseppe D'Angelo

[13] LEWG poll results for P3566R1, Sofia 2025

[14] LEWG poll results for P3655R3, Croydon 2026