The Need for Escape Hatches

Document Number:	D4035R0
Date:	2026-02-25
Audience:	LEWG

Reply-to:	Vinnie Falco vinnie.falco@gmail.com

Table of Contents

Abstract

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

Revision History

R0: February 2026 (pre-Tokyo mailing)

- Initial version.

1. 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 invariant is already established at a boundary and re-validation is pure overhead.

This pattern appears in the standard library, in production Boost libraries, and in the current `cstring_view` proposal. Three independent examples follow.

2. Standard Precedent

The standard library provides `std::condition_variable` [1], which requires `std::unique_lock<std::mutex>`. This constraint enables optimizations. When the constraint is too narrow - when the user holds a different lock type - `std::condition_variable_any` [2] provides the explicit broader path:

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

// Safe default: constrained to unique_lock<mutex>.
std::condition_variable cv;
std::unique_lock<std::mutex> lk(mtx);
cv.wait(lk);

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

The constrained interface is the default. The broader interface is explicit and named. Both exist in the standard.

3. Established Practice

Boost.URL [3] 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: invariant already established by the parser.
pct_string_view s = make_pct_string_view_unsafe(data, size, decoded_size);
```

This pattern was adopted independently by three Boost libraries: Boost.URL (`make_pct_string_view_unsafe` [4]), Boost.Process (`basic_cstring_ref` [5]), and Boost.SQLite (`cstring_ref` [6]). Three independent libraries arriving at the same design is not coincidence. It is convergence on a structural need.

4. 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)` [7] and FreeBSD `dirent.h` [8]. The common invariant for path components is that names are null-terminated and do not contain embedded null bytes POSIX Base Definitions[9]. 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 established the invariant.
        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 invariants and zero additional runtime cost.

5. Conclusion

The standard library already provides constrained defaults with explicit broader counterparts. Production libraries independently converge on the same pattern for trusted boundaries. This paper asks for no wording and no poll. It asks the committee to recognize a design value: safe by default, with explicit escape hatches where zero-cost composition requires them.

The interaction between explicit escape hatches and hardened precondition checking is a related question that deserves separate treatment.

Acknowledgements

The author thanks Jonathan Wakely, Jan Schultke, Pablo Halpern, and Nevin Liber for discussion and feedback on safe and unsafe construction paths.

References

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

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

[3] Boost.URL, https://github.com/boostorg/url

[4] Boost.URL, `pct_string_view.hpp`, https://github.com/boostorg/url/blob/develop/include/boost/url/pct_string_view.hpp

[5] Boost.Process, `cstring_ref.hpp`, https://github.com/boostorg/process/blob/develop/include/boost/process/v2/cstring_ref.hpp

[6] Boost.SQLite, `cstring_ref.hpp`, https://github.com/klemens-morgenstern/sqlite/blob/develop/include/boost/sqlite/cstring_ref.hpp

[7] FreeBSD `readdir(3)`, https://man.freebsd.org/cgi/man.cgi?query=readdir&sektion=3

[8] FreeBSD source, `sys/sys/dirent.h`, https://cgit.freebsd.org/src/tree/sys/sys/dirent.h

[9] The Open Group Base Specifications Issue 8, https://pubs.opengroup.org/onlinepubs/9799919799/