# More minor additions to standard library hardening


| | |
|---|---|
| Document #: | P4233R0 |
| Date: | 2026-05-11 |
| Project: | Programming Language C++ |
| Audience: | Library Evolution, Library |


| | |
|---|---|
| Reply-to: | Konstantin Varlamov <varconst@apple.com> |


## § 1 Introduction

This paper is a small followup to P3471R4 "Standard Library Hardening" and P3697R1 "Minor additions to C++26 standard library hardening" that proposes adding several hardened preconditions across the Library. The unifying theme is in fact a lack of a unifying theme; these are "miscellaneous" checks that don't fit into a larger category. This paper aims to work towards the goal of making sure the Library is exhaustively covered within the existing hardening paradigm.

## § 2 Revision history

— R0: Initial proposal.

## § 3 Motivation

The original hardening paper, P3471, focused on the checks that satisfied a few criteria:

— Security-critical (violating the check results in an immediate security-sensitive issue within the library implementation)

— Low overhead (in practice, this means the check is O(1))

— Widely used (so that the benefit from hardening is clear)

— Have a simple definition (it's straightforward to see the effect of hardening)

This paper aims to identify checks that "fell between the cracks" and sometimes somewhat relaxes the latter two criteria (e.g., some of the functions being hardened are not very widely used).

## § 4 A note on scope

This paper deliberately focuses on "miscellaneous" checks that don't fit neatly into a larger category. Apart from these, there are a few remaining major areas; those would require a dedicated paper or several papers:

— `<ranges>`

— `<mdspan>`, `<valarray>`, `<linalg>` (large, complex, math-related).

These major groups are explicitly out of scope for this paper.

## § 5 Summary of proposed changes

All of these checks were verified when violated to cause an OOB read or write (using Address Sanitizer) in at least one of the major implementations (with a couple of exceptions that are outlined below).

### § 5.1 `[containers]`

#### § 5.1.1 `inplace_vector`

— `unchecked_push_back`;

— `unchecked_emplace_back`.

As the name implies, these would attempt to add an element even if the underlying buffer has no capacity, resulting in an OOB write. While these functions make it clear they aim for extra performance, the other part of the motivation is likely not having to deal with error reporting – a complication that hardening sidesteps.

#### § 5.1.2 Bucket interface of unordered containers

— `b.bucket_size(n)`;

— `b.begin(n)`;

— `b.end(n)`;

— `b.cbegin(n)`;

— `b.cend(n)`.

These functions directly access the underlying buckets of an unordered container by index; an invalid index would result in an OOB access in a typical implementation. An interesting quirk is that in practice, `end` and `cend` are likely to return a sentinel iterator that doesn't depend on `n`; however, relying on that seems to be deterring too much to implementation specifics. Conceptually, there is no difference between these functions as far as wording is concerned, so it seems more intuitive and consistent to add symmetrical hardened preconditions to all of them.

— `b.bucket(k)`.

Hardening this function is more debatable compared to the functions above that directly take an index as a parameter. It has the same precondition that the bucket list must not be empty, but the function is unlikely to actually access the list in an implementation. However, it seems more straightforward to harden the precondition for consistency with the rest of the bucket interface; in practice, it is rarely used, making the performance argument somewhat moot.

### § 5.2 `[algorithms]`

(Note: where relevant, this also applies to the range versions of the algorithms and/or parallel overloads as well as overloads taking extra parameters such as comparators – omitted for brevity)

— `pop_heap(first, last)`;

— `min` and `ranges::min` (the overloads taking a range);

— `max` and `ranges::max` (the overloads taking a range);

— `minmax` and `ranges::minmax` (the overloads taking a range);

These require the range they're operating on to be non-empty as they either read from (the `minmax` group) or write to (`pop_heap`) a single element unconditionally; with an empty range that would result in dereferencing the past-the-end iterator.

— `shift_left(first, last, n)`;

— `shift_right(first, last, n)`.

A negative `n` can result in OOB reads and writes on both sides of the given range. The implementation already has to check for the `== 0` case.

### § 5.3 `[strings]`

#### § 5.3.1 `basic_string`

— `resize_and_overwrite(n, op)`.

`op` is a user-provided callable that returns the number of elements overwritten, `r`, with a precondition that `r < n`; violating the precondition likely results in an OOB access inside the library as the string updates its internal state. Checking `r <= n` is trivial.

Note that the raison d'être of this function is performance; however, for one, the overhead of a single integer comparison seems low considering this happens in the context of allocating and writing a string.