is_pointer_in_range

Document Number: **P3952 R0**
Date: 2026-01-08
Reply-to: Herb Sutter (herb.sutter@gmail.com)
Glen Joseph Fernandes (glenjofe@gmail.com)
Audience: LEWG, EWG
Follow-up to: P3234R1

Notes: The parallel proposal [P3234R1] by Glen Fernandes is merged into this paper.
The parallel proposal [P3852R0] by Hana Dusíková provides an implementation, and considers the same design decisions (we agree we should discuss whether to use `std::span`).

1 Overview

1.1 Motivation

WG21 is often asked for a way to reliably answer the question, "does this raw pointer point inside this buffer?" See [Core 2025] for some use cases. Such an `is_pointer_in_range` function is appropriate to standardize because:

- it "makes the impossible possible" in portable C++ code;
- it is already widely implemented in standard library implementations (e.g., for efficient `string` `insert`/`append`) but by relying on unspecified behavior or tolerating false positives (see §1.5 and §1.6);
- it enables more use cases that cannot tolerate false positive answers, such as checks for whether a data item is already governed by this container or is some new data that should be added/adopted, and pointer bounds checking which helps improve two vulnerability root causes perennially in the "Top 10 most dangerous software weaknesses" lists (out-of-bounds write and out-of-bounds read) [MITRE 2025];
- it is regularly requested, including by WG21 members in [P3234R1], [P3852R0], and in recurring email list discussions (including this week, [Core 2025]), and standardizing it will take less time than having the next email discussion.

1.2 Requirements

We already have "is pointer in range" that returns `true` "if" the pointer is in the range, via `std::less` (see §1.5).

We want `is_pointer_in_range` that returns `true` "if **and only if**" the pointer is in the range, which includes among other properties that it is:

- well-defined for all pointer values;
- has no false positives or false negatives; and
- works for `constexpr` pointers.

1.3 Non-requirements

Perhaps surprisingly, `is_pointer_in_range` is not directly related to having a strict total order on pointers, which the standard already supplies (see §1.5). A strict total order on pointers is **not necessary** (guarantees more than `is_pointer_in_range` needs) and **not sufficient** (doesn't guarantee enough because it doesn't prevent interleaving; see §1.5). For details, see the following subsections.

1.4 Doesn't raw pointer comparison cover this?

No. Raw pointer comparisons are only valid within the same array.

However, many programmers have used this raw pointer test to determine whether `p` points into `[b,e)`:

```
b <= p && p < e
```

This is undefined behavior unless `p`, `b`, and `e` all point into the same array, and so cannot be used to detect a `p` that does not point into an array, even if that use happens to work on most implementations.

1.5 Doesn't `std::less` (or any strict total ordering over raw pointers) cover this?

No. `std::less` and other strict total orders enable using raw pointers as associative container keys.

However, many experts (e.g., [Boost 2014]) have used this test to determine whether `p` points into `[b,e)`:

```
!std::less<T const*>{}(p, b) && std::less<T const*>{}(p, e)
```

This relies on unspecified behavior which allows false positives.

Per [comparisons.general], `std::less` is required to "yield a result consistent with the implementation-defined strict total order over pointers ([defns.order.ptr])," which is defined in [defns.order.ptr] as an "implementation-defined strict total ordering over all pointer values such that the ordering is consistent with the partial order imposed by the built-in operators `<`, `>`, `<=`, `>=`, and `<=>`."

The false-positive problem is that the standard permits a conforming implementation to use a strict total ordering in which an object `Obj` outside an array `A` could have an address `&Obj` that compares not less than `&*std::begin(A)` and also less than `&*std::end(A)`, because of pointer value interleaving such as due to segmented pointers or security-tagged pointers (not because of actual layout, since a layout that puts `Obj` actually inside `A` would be nonconforming).

1.6 Existing practice and implementations

Many projects make this a named function in some form, including large library collections such as:

- libc++: `__is_pointer_in_range` [LLVM 2025]
- Qt: `q_points_into_range` [Qt 2020]
- Boost: `pointer_in_range`, `ptr_in_range` [Boost 2014]

All of these are declared `constexpr` and are nonthrowing (though only one is declared `noexcept`).

Others write the check inline, including `std::` implementations which must implement it in order to conform:

- libstdc++: `basic_string::_M_disjunct`
- BSL: `basic_string::privateReplaceRaw`
- EASTL: `basic_string::replace`

This function also appears in static analysis tools, such as [CBMC].

The current implementations rely on techniques such as §1.5 that rely on unspecified behavior which can generate false positives, but that work in practice (pass test suites, work with tools) on the platforms they currently support. We should provide implementation help so they can be implemented portably and deterministically.

2 Discussion

2.1 Signature

This paper proposes a signature that uses the safer `span` type, because `span` should be encouraged by default and because `span` implicitly includes the requirement that this function is intended to be used only to ask whether the pointer is within a single allocation. Call sites that have three pointers can conveniently write `is_pointer_in_range(ptr, {begin, end})`, which is still almost as easy to use as a "three raw pointers" signature.

This paper proposes using the existing `span<T const>`. It was suggested in reflector discussion that `span<void>` that represents a `[begin,end)` range of `void*` could be added to the standard and used for this API, and this paper would welcome that EWG direction but does not now propose that extension because that extension would require answering significant additional design questions (for example, it's clear enough that `span<T>` should convert to `span<void>`; but `span<void>` cannot satisfy the current interface of `span` because it could not be dereferenced to return a `void&` which makes it not substitutable for the primary template; and its desirable iterator increment behavior is not obvious) and because it is less type-safe than a `span<T const>` which statically prevents pointers to mixed types.

A "three raw `void*` parameters" signature could be used instead of or in addition to the `span` signature, with Mandates/Precondition that `[begin,end)` is a valid contiguous range. However, this is considerably less type-safe and would encourage passing invalid ranges because pointers to mixed types would be harder to avoid, and because a series of parameters of the same type is known to invite call sites to get the argument order wrong without a warning (consider `std::clamp(val,lo,hi)`, `memcpy(dest,src,count)`, and `rotate(first,middle,last)`).

2.2 Naming

This paper proposes the "is-pointer" name "**is**_pointer_in_range" which follows existing practice in Boost and LLVM. Alternatively, it could be the "pointer-is" name "pointer_**is**_in_range" or just "**is**_in_range" which follows standard conventions like is_copy_assignable and is_null_pointer and is similar to existing practice in Qt.

Because the proposed interface currently uses `span`, other alternatives are "is_pointer_in_span" or "span::contains".

2.3 Implementability

Hana Dusíková has implemented this proposal with a Clang compiler intrinsic, as described in her parallel proposal [P3852R0]. Her implementation is as follows, using an "are pointers related" intrinsic:

```
_LIBCPP_HIDE_FROM_ABI constexpr bool is_pointer_between(
  const void * __ptr,
  const void * __first,
  const void * __last
) {
  if consteval {
    _LIBCPP_ASSERT_UNCATEGORIZED(__builtin_pointers_related(__first, __last),
      "Pointers __first and __last must be pointing to same top-level object.");
    if (!__builtin_pointers_related(__first, __ptr)) {
      return false;
    }
  }
  return (__first <= __ptr) && (__ptr < __last);
}
```

On the large majority of platforms where the `std::less` technique does not have false positives, a conforming implementation could use the following Boost-style implementation. (Implementations can be cleverer; this is only to document a baseline capability.)

```
template<class T>
constexpr bool is_pointer_in_range(T const* ptr, std::span<T const> s) {
  if (std::is_constant_evaluated()) {
    for (auto i = 0; i < std::ssize(s); ++i)
      if (ptr == &s[i])
          return true;
    return false;
  }
  return !std::less<>{}(ptr, &*s.begin()) && std::less<>{}(ptr, &*s.end());
}
```

3 Wording

Insert into [memory.syn] after `to_address`:

```
template<typename T>
constexpr bool is_pointer_in_range(T const* p, std::span<T const> s) noexcept;
```

Insert after [pointer.conversion]:

x.x.x. Pointer range check [pointer.range.check]

```
template<typename T>
constexpr bool is_pointer_in_range(T const* p, std::span<T const> s) noexcept;
```

Mandates: T is not a function type.

Returns: `true` if `p == &s[i]` for some unsigned integer `i` where `0 <= i && i < std::size(s)`, otherwise `false`.

Recommended practice: Implementations should be O(1) on platforms where possible.

4 Acknowledgments

Thanks to Glen Fernandes for [Boost 2014] and [P3234R1] to which this paper is a successor.

Thanks to Hana Dusíková for the parallel paper [P3852R0] published two weeks before this one, with my (Herb's) apologies for not seeing it, and for her Clang prototype implementation demonstrating feasibility.

Thanks to Gašper Ažman, Ben Boeckel, Marshall Clow, Peter Dimov, Gabriel Dos Reis, Hana Dusíková, Daniela Engert, Ion Gaztañaga, Howard Hinnant, Oliver Hunt, Tomasz Kamiński, Nikolas Klauser, Nevin Liber, Jens Maurer, Ian Petersen, Barry Revzin, Richard Smith, Tim Song, Lénárd Szolnoki, Daveed Vandevoorde, Tony Van Eerd, Ville Voutilainen, Jonathan Wakely, Jarrad Waterloo, and everyone else who participated in [Core 2025].

5 References

[Boost 2014] G. Fernandes. “`pointer_in_range`” (Boost, 2014).

[CBMC] CBMC function Pointer_in_range (GitHub).

[Core 2025] “Why is three-way comparison of pointers defined as `strong_ordered`?” (WG21 CWG mailing list, December 2025).

[LLVM 2025] LLVM file is_pointer_in_range.h (GitHub, 2025).

[MITRE 2025] “2025 CWE Top 25 Most Dangerous Software Weaknesses” (mitre.org, 2025).

[P3234R1] G. Fernandes. “Utility to check is a pointer is in a given range” (WG21 paper, 2024).

[P3852R0] H. Dusíková. “A (constexpr) utility to check if pointer points between two related pointers” (WG21 paper, 2025).

[Qt 2020] Qt file qcontainertools_impl.h (Qt.io, 2020).