

Add a `std::priority_queue` method to move the top element from it

Document #:
Date: 2026-09-28
Project: Programming Language C++
Audience: LEWG
Reply-to: Aleksei Sidorin
<alexey.v.sidorin@yandex.ru>

Contents

1	Introduction	1
2	Motivation and Scope	1
3	Impact On the Standard	2
4	Design decisions and notes	2
4.1	Freestanding functions	2
4.2	Exception safety	3
4.3	Interaction with <code>trivial_relocatability</code>	3
4.4	Naming	3
5	Benchmarking and testing	3
5.1	<code>priority_queue::pop()</code> code size: <code>pop_heap()</code> vs <code>remove_heap()</code>	4
5.2	<code>priority_queue::displace_top()</code> code size vs <code>priority_queue::top()</code> and <code>pop()</code>	4
6	Proposed wording	5
7	Implementation experience	7
8	Acknowledgments	7
9	Related work	7

1 Introduction

Unlike other containers, `std::priority_queue` doesn't allow its users to move elements out of it. This asymmetry causes performance hits and limits the type usage. I propose adding a corresponding method to `std::priority_queue` backed by a set of freestanding functions in `<algorithm>` header.

2 Motivation and Scope

In order to extract the top element from `std::priority_queue`, STL users have to write code similar to the following:

```
auto x = queue.top(); // Copy constructor or copy assignment.  
queue.pop();
```

Unfortunately, we cannot move the top value from the queue because `top()` is of `const_reference` type (23.6.4.4 Members [[prqueue.members](#)]).

The first problem is that this can cause a performance hit if the value type is expensive to copy (a structure containing large strings, allocating copy constructors, etc.).

The second problem is that this also makes impossible to use `std::priority_queue` with move-only types like `std::unique_ptr`.

I've made a search in chromium sources showing that almost all heap structure usage have a similar pattern:

```
std::pop_heap(queue.begin(), queue.end());
*event = std::move(queue.back());
queue.pop_back();
```

(https://github.com/search?q=repo%3Achromium%2Fchromium%20pop_heap&type=code)

I think this clearly indicates that STL can have a better interface here.

Another issue with `top()` & `pop()` approach is that `pop()` uses `std::pop_heap()` + `c.pop_back()` call pair internally. And `std::pop_heap()` is required to move the top element to the sequence end. This introduces stores hard to elide by compilers. For example, in <https://godbolt.org/z/r7KK7MYzc> we can find such a store in assembly line 13 left even with `-O3`.

Therefore, I propose adding an `std::priority_queue` method behaving like pseudo-code below:

```
T displace_top() {
    T val = std::move_if_noexcept(container.front());
    pop();
    return val;
}
```

but preserving the heap invariant.

In addition, I propose adding freestanding `<algorithm>` functions:

- `std::displace_heap()` and `std::ranges::displace_heap()` will return the queue top value and remove it from the queue with restoring heap property of the data structure. Unlike `std::pop_heap()`, these functions leave the last range element in a moved-from state, so handling range length after calling them lies on users. These functions will not require placing the previous top element to the container end, saving one move assignment call.
- `std::remove_heap()` and `std::ranges::remove_heap()` behave like `displace_heap()`, but return `void` and just discard the top element without returning it.

With these functions, we can also save one move assignment in `std::priority_queue::pop()` by changing its effect from `std::pop_heap()` to `std::remove_heap()`.

3 Impact On the Standard

The change is a pure library extension which can be implemented with existing language features and current C++ compilers.

4 Design decisions and notes

4.1 Freestanding functions

As Arthur O'Dwyer correctly pointed, it may be not evident at first glance why do we need additional freestanding functions to support `priority_queue::displace_top()` in this proposal.

The first reason is symmetry: `pop()` and `push()` have their `pop_heap()` and `push_heap()` twins in `<algorithm>`, so reflecting a new `priority_queue` method in `<algorithm>` also seems to be a good idea.

The second reason is effectiveness. The Standard requires `std::priority_queue::pop()` to behave like `std::pop_heap()` and `c.pop_back()` pair. And that's exactly how it is implemented in `libc++` and `libstdc++`. The modern STL implementations, like those used in `libc++` and `libstdc++`, implement `pop_heap()` not in a classical "swap-and-sift-up" manner, but using Floyd algorithm: we store the top value in a temporary variable, then `sift_up` the result hole and put the saved value into the last position. So, `pop_heap()` requires dedicated instructions (a move assignment) to place the top element to the container end. This final move assignment is not needed if we need to get the old top immediately, which is a very frequent usage scenario for heap. For `priority_queue::pop()` implementation, this move assignment is

redundant as well, because the following `pop_back()` just immediately destroys the value. That's why we need functions behaving differently from `pop_heap()`: in order to implement `pop()` effectively, we need a way to discard the popped value without performing additional operations with it. See [Benchmarking and testing](#) for measurements and explanation.

For freestanding functions, the pattern `pop_heap()` immediately followed by `pop_back()` can be met pretty frequently, as Github search shows: https://github.com/search?q=std%253A%253Apop_heap&type=code. This justifies adding `remove_heap()` separately instead of making it a `priority_queue::pop()` implementation detail.

4.2 Exception safety

A common discussion point for this proposal predecessors was exception safety. This paper proposes to apply `move_if_noexcept` behaviour when extracting the top value, so the data structure will keep its consistent state even if move constructor can throw.

It is often pointed that the `priority_queue` internal bookkeeping also involves non-trivial operations, but this proposal does not affect this maintenance, so we do not discuss it.

4.3 Interaction with trivial_relocatability

There were several proposals introducing `trivial_relocatability` concept:

- <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p1144r13.html>
- <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p2786r13.html>
- <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2026/p4102r0.html>

This proposal stacks with trivial relocatability concept pretty well, because we can use element relocation instead of moving, and this will save even more work. For example, we can relocate the first element into the result in `displace_heap()`, and then relocate elements while performing sift operations. `priority_queue::pop()` can also use something like `pop_back_relocated()` of the underlying container to avoid calling an unneeded destructor.

The primary difference is that if trivial relocatability is used, the freestanding functions will require to leave the last element not in a moved-from, but in a relocated-from state. Freestanding functions can maintain the current behaviour for safety (leaving the last element as moved-from), while `priority_queue` can exploit all concept abilities.

4.4 Naming

The naming can be a subject for discussion. Below are the reasons why the current naming has been chosen.

- `displace` verb was chosen as an antonym for `emplace`. `emplace` constructs a new element, and `displace` (conceptually) destroys that element, returning its value. Another option is `extract`, because we have `std::map::extract()` in the STL. But `map`'s `extract()` returns a handle, not an element, so it describes the action worse.
- `remove` had multiple candidate verbs. Actually, name `pop_heap` is better for the function named `remove_heap()` in this proposal, because `pop_back()` and `pop_front()` do not store the removed element and just discard it, as `remove_heap()` does. Also, its behaviour is very similar to `priority_queue::pop()`. However, this name is already in use, so I stick to `remove` which is similar to `std::remove()`, because it leaves the last element in a moved-from state too. This makes it a better candidate than `erase_heap()`. `shift_left()` rather describes moving of continuous ranges rather than heap internal machinery.

5 Benchmarking and testing

When talking about saving a single move assignment in a $O(\log(n))$ algorithm, there is no much difference in the run time, at least with modern compilers. My time measuring benchmarks (based on libc++ internal benchmarking system) have shown a noise-like difference, even though the instruction count in the updated versions was less. However, there is a clear win in the generated code size, and this win is stable across measurements with different types. It even allowed additional function inlining for some complex types.

5.1 `priority_queue::pop()` code size: `pop_heap()` vs `remove_heap()`

In this benchmark, `std::priority_queue::pop()` was implemented with `pop_heap()` vs `remove_heap()`. The benchmark compiles the code below with different types and measures the generated machine code size for x86_64 and AArch64.

```
template <class T>
__attribute__((noinline)) void test_pq_pop(std::priority_queue<T>& pq) {
    pq.pop();
}
```

Type	Current, x86_64	Proposed, x86_64	Delta	Current, AArch64	Proposed, AArch64	Delta
<code>uint32_t</code>	260 B	237 B	-23 B (-8.8%)	260 B	232 B	-28 B (-10.8%)
<code>uint64_t</code>	262 B	239 B	-23 B (-8.8%)	260 B	232 B	-28 B (-10.8%)
<code>float</code>	238 B	238 B	0 (0.0%) ¹	244 B	232 B	-12 B (-4.9%)
<code>pair<u32,u32></code> ²	414 B	391 B	-23 B (-5.6%)	388 B	324 B	-64 B (-16.5%)
<code>tuple<u32,u64,u32></code>	545 B	480 B	-65 B (-11.9%)	612 B	552 B	-60 B (-9.8%)
<code>string</code> ²	1118 B	914 B	-204 B (-18.2%)	1036 B	876 B	-160 B (-15.4%)

With aggressive inlining enabled (`-mllvm -inline-threshold=100000`, no non-inlined functions left):

Type	Current, x86_64	Proposed, x86_64	Difference	Current, AArch64	Proposed, AArch64	Delta
<code>uint32_t</code>	260 B	237 B	-23 B (-8.8%)	260 B	232 B	-28 B (-10.8%)
<code>uint64_t</code>	262 B	239 B	-23 B (-8.8%)	260 B	232 B	-28 B (-10.8%)
<code>float</code>	238 B	238 B	0 (0.0%)	244 B	232 B	-12 B (-4.9%)
<code>pair<u32,u32></code>	417 B	391 B	-26 B (-6.2%)	352 B	320 B	-32 B (-9.1%)
<code>tuple<u32,u64,u32></code>	520 B	460 B	-60 B (-11.5%)	604 B	512 B	-92 B (-15.2%)
<code>string</code>	887 B	800 B	-87 B (-9.8%)	744 B	720 B	-24 B (-3.2%)

¹ The instruction size diff was compensated by alignment NOPs.

² If some function was inlined in one case and not in another, we add the non-inlined function size to the total size.

5.2 `priority_queue::displace_top()` code size vs `priority_queue::top()` and `pop()`

The test is the compilation of the code below with different types and measuring the generated code size. The results for `test_original_top_then_pop()` are given for the initial version.

```
template <class T>
__attribute__((noinline)) T test_original_top_then_pop(std::priority_queue<T>& pq) {
    T t = pq.top();
}
```

```

pq.pop();
return t;
}

template <class T>
__attribute__((noinline)) T test_displace_top(std::priority_queue<T>& pq) {
    return pq.displace_top();
}

```

Type	displace_top, x86_64	top + pop, x86_64	Delta	displace_top, AArch64	top + pop, AArch64	Delta
uint32_t	260 B	256 B	-4 B (-1.5%)	264 B	240 B	-24 B (-9.1%)
uint64_t	262 B	257 B	-5 B (-1.9%)	264 B	240 B	-24 B (-9.1%)
float	252 B	238 B	-14 B (-5.6%)	260 B	236 B	-24 B (-9.2%)
pair<u32,u32>	429 B	396 B	-33 B (-7.7%)	404 B	332 B	-72 B (-17.8%)
tuple<u32,u64,u32>	567 B	502 B	-65 B (-11.5%)	628 B	568 B	-60 B (-9.6%)
string	1434 B	1006 B	-428 B (-29.8%)	1408 B	952 B	-456 B (-32.4%)

With aggressive inlining enabled (`-mllvm -inline-threshold=100000`, no non-inlined functions left):

Type	displace_top, x86_64	top + pop, x86_64	Delta	displace_top, AArch64	top + pop, AArch64	Delta
uint32_t	260 B	256 B	-4 B (-1.5%)	264 B	240 B	-24 B (-9.1%)
uint64_t	262 B	257 B	-5 B (-1.9%)	264 B	240 B	-24 B (-9.1%)
float	252 B	238 B	-14 B (-5.6%)	260 B	236 B	-24 B (-9.2%)
pair<u32,u32>	420 B	396 B	-24 B (-5.7%)	356 B	328 B	-28 B (-7.9%)
tuple<u32,u64,u32>	555 B	492 B	-63 B (-11.3%)	620 B	528 B	-92 B (-14.8%)
string	934 B	840 B	-94 B (-10.1%)	804 B	708 B	-96 B (-11.9%)

The inlining accounting is the same as in the previous benchmark.

6 Proposed wording

Add an entry into chapter 26.8.8 Heap operations [\[alg.heap.operations\]](#): `[displace_heap.heap]`

```

template <class RandomAccessIterator>
constexpr iter_value_t<RandomAccessIterator>
    displace_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
constexpr iter_value_t<RandomAccessIterator>
    displace_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

```

```

template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
        class Proj = identity>
requires sortable<I, Comp, Proj>
constexpr iter_value_t<I>
    ranges::displace_heap(I first, S last, Comp comp = {}, Proj proj = {});

template<random_access_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr iter_value_t<iterator_t<R>>
    ranges::displace_heap(R&& r, Comp comp = {}, Proj proj = {});

```

1. Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.
2. *Preconditions:* The range `[first, last)` is a valid non-empty heap with respect to `comp` and `proj`. For the overloads in namespace `std`, `RandomAccessIterator` meets the *Cpp17ValueSwappable* requirements (16.4.4.3) and the type of `*first` meets the *Cpp17MoveConstructible* (Table 31) and *Cpp17MoveAssignable* (Table 33) requirements.
3. *Effects:* Replaces the value in the location `first` with the value in the location `last - 1` and makes `[first, last - 1)` into a heap with respect to `comp` and `proj`.
4. *Returns:* the initial value in the location `first`.
5. *Complexity:* At most $2 \log(\text{last} - \text{first})$ comparisons and twice as many projections.

Add an entry into chapter 26.8.8 Heap operations [\[alg.heap.operations\]](#): [\[remove_heap.heap\]](#)

```

template <class RandomAccessIterator>
constexpr void
    remove_heap(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
constexpr void
    remove_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
        class Proj = identity>
requires sortable<I, Comp, Proj>
constexpr void
    ranges::remove_heap(I first, S last, Comp comp = {}, Proj proj = {});

template<random_access_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr void
    ranges::remove_heap(R&& r, Comp comp = {}, Proj proj = {});

```

1. Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.
2. *Preconditions:* The range `[first, last)` is a valid non-empty heap with respect to `comp` and `proj`. For the overloads in namespace `std`, `RandomAccessIterator` meets the *Cpp17ValueSwappable* requirements (16.4.4.3) and the type of `*first` meets the *Cpp17MoveConstructible* (Table 31) and *Cpp17MoveAssignable* (Table 33) requirements.
3. *Effects:* Replaces the value in the location `first` with the value in the location `last - 1` and makes `[first, last - 1)` into a heap with respect to `comp` and `proj`.
4. *Returns:* `last` for the overloads in namespace `ranges`.
5. *Complexity:* At most $2 \log(\text{last} - \text{first})$ comparisons and twice as many projections.

Add an entry into chapter 23.6.4.4 Members [\[priqueue.members\]](#):

```
T displace_top();
```

Effects: As if by:

```

auto result = displace_heap(c.begin(), c.end(), comp);
c.pop_back();
return result;

```

Change the wording in 23.6.4.4 Members [\[priqueue.members\]](#) (`priority_queue::pop()`):

Effects: As if by:

```
remove_heap(c.begin(), c.end(), comp);
```

```
pop_heap(c.begin(), c.end(), comp);
```

```
c.pop_back();
```

7 Implementation experience

I made a reference implementation of the proposed additions in my `libc++` fork. The diff is available as a [Github pull request](#) and a [branch in my repository copy](#).

8 Acknowledgments

Thanks to:

- Anton Polukhin for initial proposal proof-read
- Arthur O'Dwyer for suggesting `move_if_noexcept` and contributing a lot of ideas to this proposal

9 Related work

<https://lists.isocpp.org/std-proposals/2021/02/2390.php> - a similar proposal. No actions were taken after publishing it to std-proposals.

<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3182r1.html> proposes adding `pop_value` and `push_value` methods to many STL containers. Unfortunately, the author (Brian Bi) confirmed that it is stalled.