Add std::priority queue method to move the top element from it

Document #:
Date: 2026-03-08
Project: Programming Language C++
Audience: LEWG
Reply-to: Aleksei Sidorin
<alexey.v.sidorin@yandex.ru>

Contents

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 [priqueue.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 can introduce 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 an unspecified 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 `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

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()`.

— 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++. `pop_heap()`, in part, requires dedicated instructions (a move assignment) to place the top element to the container end in addition to `sift_up`. This 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 Benchmarks for performance hit measurement and explanation.

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.

5 Technical specification

Add an entry into chapter 26.8.8 [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(`last - first`) comparisons and twice as many projections.

Add an entry into chapter 26.8.8 [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(`last - first`) comparisons and twice as many projections.

Add an entry into chapter 23.6.4.4 [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 [priqueue.members] (`priority_heap::pop()`):

Effects: As if by:

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

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

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

6 Reference implementation

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

7 Benchmarks

I compared the performance of `pop_heap()` + `pop_back()` pair used to implement `priority_queue::pop()` against `remove_heap()` + `pop_back()` suggested in this proposal. In order to perform the measurement, I used a modified `pop_heap.bench.cpp` benchmark from libc++ test suite. It generates heap-sorted vectors of different fixed types and size and performs `pop_heap()` for all vector elements. The benchmark has shown a small, but stable performance win for `remove_heap()` version, showing 2-5% speed improvement for most types, and even 20% for `float` version.

1. For `uint32_t` and `uint64_t` moving from `pop_heap() + pop_back()` removes two instructions which are stores to the last element in two different execution branches. It also saves one register, because we can avoid keeping the top value for the entire function execution time, so this register isn't pushed/popped in the preamble and before returning. This result is expected. We remove 4 instructions of 88 (including alignment nops). The execution time for both versions doesn't show a notable difference.
2. For `pair<uint32_t, uint32_t>` the compiler has managed to inline `remove_heap()` call, but didn't inline `pop_heap()`. The original version with `pop_heap` was not inlined. This can explain the notable speedup shown in the benchmark. I guess the generated `pop_heap()` has exceed the compiler inlining threshold. This reminds us that even a small operation count difference can affect the performance a lot (10.8% improvement).
3. For `tuple<uint32_t, uint64_t, uint32_t>` the compiler didn't inline both functions, but `pop_heap()` has more instructions due to spills, because it tries to keep the tuple elements in registers, and the three registers storing the top value are making the register set insufficient. The measured speedup is ~5%.
4. For `std::string`, `remove_heap()` was inlined, but `pop_heap()` wasn't. However, `sift_up()` was inlined into `pop_heap()`, but left non-inlined while inlining `remove_heap()`. Unlike trivial types, moving from the last string requires writing to it, but the top value kept for the entire `pop_heap()` still consumes registers. The speedup is small but well-reproducible (~2%).
5. For `float`, the speedup is the largest: ~20.7%, even though the instruction diff is just two instructions.

7.1 `pop_heap() + pop_back()` benchmark results

```
Run on (7 X 2600 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x7)
  L1 Instruction 32 KiB (x7)
  L2 Unified 256 KiB (x7)
  L3 Unified 6144 KiB (x1)
Load Average: 1.20, 0.87, 0.94
```

Benchmark	Time	CPU	Iterations
uint32_1	3.51 ns	3.51 ns	199491584
uint32_4	3.73 ns	3.73 ns	187170816
uint32_16	8.65 ns	8.65 ns	81002496
uint32_64	15.1 ns	15.1 ns	46661632
uint32_256	19.9 ns	19.9 ns	34865152
uint32_1024	29.0 ns	29.0 ns	24117248
uint32_16384	46.3 ns	46.3 ns	15204352
uint32_262144	63.6 ns	63.6 ns	11010048
uint32_4194304	79.1 ns	79.1 ns	12582912
uint64_1	3.99 ns	3.99 ns	174587904
uint64_4	3.88 ns	3.88 ns	180617216
uint64_16	9.61 ns	9.61 ns	72876032
uint64_64	14.9 ns	14.9 ns	47185920
uint64_256	19.9 ns	19.9 ns	34078720
uint64_1024	29.5 ns	29.4 ns	23592960
uint64_16384	46.3 ns	46.3 ns	15204352
uint64_262144	63.6 ns	63.6 ns	11010048
uint64_4194304	80.2 ns	80.2 ns	12582912
pair<uint32, uint32>_1	4.59 ns	4.59 ns	152305664
pair<uint32, uint32>_4	6.40 ns	6.39 ns	109314048
pair<uint32, uint32>_16	11.7 ns	11.7 ns	60030976

```
pair<uint32, uint32>_64                    16.6 ns         16.6 ns     42205184
pair<uint32, uint32>_256                   23.9 ns         23.9 ns     27262976
pair<uint32, uint32>_1024                  69.7 ns         69.7 ns     10223616
pair<uint32, uint32>_16384                 93.2 ns         93.2 ns      7602176
pair<uint32, uint32>_262144                 110 ns          110 ns      6553600
pair<uint32, uint32>_4194304                130 ns          130 ns      4194304
tuple<uint32, uint64, uint32>_1            5.82 ns         5.81 ns    120061952
tuple<uint32, uint64, uint32>_4            7.46 ns         7.46 ns     93847552
tuple<uint32, uint64, uint32>_16           11.0 ns         10.9 ns     63963136
tuple<uint32, uint64, uint32>_64           16.8 ns         16.8 ns     41680896
tuple<uint32, uint64, uint32>_256          22.4 ns         22.4 ns     28835840
tuple<uint32, uint64, uint32>_1024         72.0 ns         72.0 ns      9961472
tuple<uint32, uint64, uint32>_16384         106 ns          106 ns      6815744
tuple<uint32, uint64, uint32>_262144        125 ns          125 ns      5767168
tuple<uint32, uint64, uint32>_4194304       166 ns          166 ns      4194304
string_1                                   33.7 ns         33.7 ns     20971520
string_4                                   49.4 ns         49.4 ns     14155776
string_16                                  82.1 ns         82.0 ns      8650752
string_64                                   122 ns          122 ns      5767168
string_256                                  159 ns          159 ns      4456448
string_1024                                 202 ns          202 ns      3670016
string_16384                                344 ns          344 ns      2097152
string_262144                               704 ns          704 ns      1048576
string_4194304                             1259 ns         1258 ns      4194304
float_1                                    3.49 ns         3.49 ns    201064448
float_4                                    4.34 ns         4.34 ns    161480704
float_16                                   6.64 ns         6.64 ns    105644032
float_64                                   9.69 ns         9.69 ns     72351744
float_256                                  13.1 ns         13.1 ns     53477376
float_1024                                 53.4 ns         53.4 ns     13107200
float_16384                                68.6 ns         68.6 ns     10223616
float_262144                               79.3 ns         79.3 ns      8912896
float_4194304                              85.0 ns         85.0 ns      8388608
```

7.2 `remove_heap()` + `pop_back()` benchmark results

```
Run on (7 X 2600 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x7)
  L1 Instruction 32 KiB (x7)
  L2 Unified 256 KiB (x7)
  L3 Unified 6144 KiB (x1)
Load Average: 0.13, 0.98, 1.36
-------------------------------------------------------------------------------
Benchmark                                     Time             CPU   Iterations
-------------------------------------------------------------------------------
uint32_1                                   3.61 ns         3.60 ns    193462272
uint32_4                                   3.56 ns         3.56 ns    196870144
uint32_16                                  8.57 ns         8.57 ns     81788928
uint32_64                                  16.8 ns         16.8 ns     41943040
uint32_256                                 19.7 ns         19.7 ns     35389440
uint32_1024                                29.5 ns         29.5 ns     23592960
uint32_16384                               46.6 ns         46.6 ns     15204352
```

```
uint32_262144                              63.6 ns         63.6 ns     11010048
uint32_4194304                             79.0 ns         79.0 ns     12582912
uint64_1                                   3.47 ns         3.47 ns    199753728
uint64_4                                   3.59 ns         3.58 ns    194772992
uint64_16                                  8.77 ns         8.77 ns     79953920
uint64_64                                  17.0 ns         17.0 ns     41418752
uint64_256                                 19.9 ns         19.9 ns     34340864
uint64_1024                                29.4 ns         29.3 ns     23592960
uint64_16384                               46.1 ns         46.1 ns     15204352
uint64_262144                              63.9 ns         63.8 ns     11010048
uint64_4194304                             79.9 ns         79.9 ns     12582912
pair<uint32, uint32>_1                     4.20 ns         4.20 ns    166461440
pair<uint32, uint32>_4                     4.42 ns         4.42 ns    158072832
pair<uint32, uint32>_16                    9.54 ns         9.53 ns     73400320
pair<uint32, uint32>_64                    14.3 ns         14.3 ns     48758784
pair<uint32, uint32>_256                   20.1 ns         20.1 ns     33816576
pair<uint32, uint32>_1024                  62.8 ns         62.8 ns     11272192
pair<uint32, uint32>_16384                 86.0 ns         86.0 ns      8388608
pair<uint32, uint32>_262144                97.6 ns         97.6 ns      7340032
pair<uint32, uint32>_4194304                116 ns          116 ns      8388608
tuple<uint32, uint64, uint32>_1            5.55 ns         5.55 ns    125566976
tuple<uint32, uint64, uint32>_4            6.21 ns         6.20 ns    112459776
tuple<uint32, uint64, uint32>_16           9.57 ns         9.57 ns     72876032
tuple<uint32, uint64, uint32>_64           15.2 ns         15.2 ns     45875200
tuple<uint32, uint64, uint32>_256          20.9 ns         20.9 ns     32243712
tuple<uint32, uint64, uint32>_1024         67.8 ns         67.8 ns     10485760
tuple<uint32, uint64, uint32>_16384        96.6 ns         96.5 ns      7340032
tuple<uint32, uint64, uint32>_262144        115 ns          115 ns      6291456
tuple<uint32, uint64, uint32>_4194304       158 ns          158 ns      4194304
string_1                                   31.6 ns         31.6 ns     23068672
string_4                                   47.2 ns         47.2 ns     14680064
string_16                                  78.8 ns         78.8 ns      9175040
string_64                                   116 ns          116 ns      6291456
string_256                                  150 ns          150 ns      4718592
string_1024                                 189 ns          189 ns      3932160
string_16384                                324 ns          324 ns      2359296
string_262144                               683 ns          683 ns      1048576
string_4194304                             1234 ns         1234 ns      4194304
float_1                                    4.19 ns         4.19 ns    167510016
float_4                                    3.98 ns         3.98 ns    175636480
float_16                                   5.08 ns         5.08 ns    137887744
float_64                                   7.63 ns         7.63 ns     92012544
float_256                                  10.4 ns         10.4 ns     67108864
float_1024                                 36.2 ns         36.2 ns     19136512
float_16384                                57.6 ns         57.6 ns     12320768
float_262144                               59.0 ns         59.0 ns     12058624
float_4194304                              63.4 ns         63.3 ns     12582912
```

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.