Document number: PXXXX

Date: 2015-09-24

Project: The C++ Programming Language, Core Working Group

Title: Improving the standard library sorting algorithms

Revises:

Reply-to: Thibaut Le Jehan ⟨ lejehan.thibaut@gmail.com ⟩

Table of Contents

I Introduction

This paper proposes to extend `std::sort` and `std::stable_sort` so that they can work with forward and bidirectional iterators. We first analyze the current state of sorting in the standard library, then have a look at existing sorting algorithms and how they can be implemented to work reasonably efficiently with forward and bidirectional iterators. We then propose complexity guarantees for the new versions of `std::sort` and `std::stable_sort`.

II Motivation and Scope

Currently, the functions `std::sort` and `std::stable_sort` only work with random-access iterators, making it possible to sort classes such as `std::vector`, `std::deque` and `std::string`. However, the standard library also has the container classes `std::list` and `std::forward_list` which respectively provide bidirectional and forward iterators. Both of them has a member function `sort` to perform the job though, but having such a member function is hardly generic. Even the recent C++ Core Guidelines [1] agree on that:

> It is probably a dumb idea to define a `sort` as a member function of a container, but it is not unheard of and it makes a good example of what not to do.

`std::list::sort` and `std::forward_list::sort` being member functions probably has to do with the fact that these algorithms can make more assumptions knowing that they work on list data structures, allowing for a $\mathcal{O}(\log n)$ space complexity instead of a $\mathcal{O}(n)$ one for a classic mergesort. However, this approach is hardly generic: it only allows to sort full lists instead of arbitrary pairs of iterators, and it forces to add a corresponding member function `sort` to custom classes wrapping a list and offering an iterator interface.

To add to the confusion, the standard mandates that the member functions `std::list::sort` and `std::forward_list::sort` are stable even though the name does not reflect the guarantee (while `std::stable_sort` makes it clear).

Basically, we would like to be able to sort pairs forward, bidirectional and random-access iterators through a common algorithm interface.

III Discussion

Complexity of `std::sort`

Some versions of quicksort work reasonably well with forward iterators with an average $\mathcal{O}(n \log n)$ complexity but have a $\mathcal{O}(n^2)$ worst case complexity. A simple mergesort may always have a $\mathcal{O}(n \log n)$ complexity, but it requires $\mathcal{O}(n)$ auxiliary memory. The most efficient sorting algorithms such as introsort [2] and patter-defeating quicksort [3] only work with random-access iterators.

We think that the right approach to this problem is to only guarantee an $\mathcal{O}(n \log n)$ average complexity for forward and bidirectional iterators, and no worst case complexity. This is how it was done in C++03 for random-access iterators, before efficient algorithms with a $\mathcal{O}(n \log n)$ worst case complexity were discovered. A mergesort could have a $\mathcal{O}(n \log n)$ worst case complexity for bidirectional iterators, but it requires additional memory and `std::sort` has an history of not using much additional memory to sort a collection.

Complexity of `std::stable_sort`

The current complexity mandated for `std::stable_sort` is described as follows:

> It does at most $N \log^2(N)$ (where N `== last - first`) comparisons; if enough extra memory is available, it is $N \log(N)$.

This description more or less corresponds to a trivial mergesort implementation using `std::inplace_merge` to perform the merge operation. Since `std::inplace_merge` accepts bidirectional iterators, it is easy to make `std::stable_sort` work with bidirectional iterators as well, with the exact same complexity guarantees as random-access iterators.

In-place merge algorithms also exist for forward iterators, with or without using a temporary buffer, even though the standard library function `std::inplace_merge` is only guaranteed to work with bidirectional iterators. It should be possible to write a mergesort algorithm for forward iterators using such a function, leading to an $\mathcal{O}(n \log n)$ stable sort for forward iterators when enough extra memory is available.

TODO: what would be the complexity of `std::inplace_merge` and `std::stable_sort` for forward iterators when no extra memory is available?

Benefits of `sort` member functions

`std::list::sort` and `std::forward_list::sort` can be more efficient than general-purpose stable sorting algorithms for forward and bidirectional iterators; this is due to the fact that they take advantage of the specific structure of the list (they can rebind nodes). One could say that having general-purpose sorting algorithms for bidirectional and forward iterators might encourage to use them on these data structures while they might be less efficient than the member functions.

That said, libc++ at least implements these functions with `static` helper functions which take an arbitrary pair of iterators to sort. We consider this a good enough proof that the list sorting functions can be implemented as free functions taking an arbitrary pair of list iterators. That means that implementations should be allowed to overload `std::stable_sort` for pairs of list iterators under the *as-if* rule so that sorting lists with it is as efficient as using the `sort` member functions.

To sum up, good implementations can make sure that sorting a standard collection with `std::sort` or `std::stable_sort` is always as efficient as sorting it with a `sort` member function. We can have both efficiency and genericity.

Parallel sorting algorithms

This proposal focuses on the sequential algorithms from `<algorithm>` but the sorting parallel algorithms from the Parallelism TS [4] could be another target for such a change. We do not propose such changes, but encourage people to check whether it can be done and which would be the required algorithmic complexities.

Range algorithms

The Ranges TS [5] is yet another obvious target for such a change. It currently redefines `std::sort` and `std::stable_sort` so that they can work with an `[iterator, sentinel)` pair or a full iterable.

No change was made to the required iterator category, random-access iterators are still needed everywhere. Our proposal and the ranges one may be considered at the same time, in which case, we also propose to make the range-based `std::sort` and `std::stable_sort` use the `ForwardIterator` concept instead of the `RandomAccessIterator` one and the `ForwardIterable` concept instead of the `RandomAccessIterable` one.

IV Proposed wording

25.4.1.1 `sort` [sort]

```
template<class ForwardIterator>
  void sort(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  void sort(ForwardIterator first, ForwardIterator last,
            Compare comp);
```

1 *Effects:* Sorts the elements in the range `[first, last)`.

2 *Requires:* `ForwardIterator` shall satisfy the requirements of `ValueSwappable` (17.6.3.2). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

3 *Complexity:* $\mathcal{O}(N \log N)$ (where N `== last - first`) comparisons on average. $\mathcal{O}(N \log N)$ if `ForwardIterator` additionally satisfies the requirements of `RandomAccessIterator`.

25.4.1.1 `stable_sort` [stable.sort]

```
template<class ForwardIterator>
  void stable_sort(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  void stable_sort(ForwardIterator first, ForwardIterator last,
                   Compare comp);
```

4 *Effects:* Sorts the elements in the range `[first, last)`.

5 *Requires:* `ForwardIterator` shall satisfy the requirements of `ValueSwappable` (17.6.3.2). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

6 *Complexity:* It does at most $N \log^2(N)$ (where N `== last - first`) comparisons; if enough extra memory is available, it is $N \log(N)$.

7 *Remarks:* Stable (17.6.5.7).

V Conclusion

VI Acknowledgments

Bibliography

[1] B. Stroustrup and H. Sutter. C++ core guidelines. [Online]. Available: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

[2] Wikipedia, "Introsort — wikipedia, the free encyclopedia," 2015. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Introsort&oldid=680161322

[3] O. Peters. Pattern-defeating quicksort. [Online]. Available: https://github.com/orlp/pdqsort

[4] J. Hoberock. Programming languages technical specification for c++ extensions for parallelism. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507.pdf

[5] E. Niebler. Working draft, c++ extensions for range. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4382.pdf