Document number: PXXXX

Date: 2016-02-05

Project: The C++ Programming Language, Library Evolution
Working Group

Title: Weakening the iterator categories of some standard algorithms

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

Table of Contents

I Introduction

This paper proposes to weaken the iterator categories of several standard library generic algorithms to reflect the advances in made in the realm of algorithms. Such changes were already proposed in the Palo Alto report [1] but were later dropped in the Ranges TS [2], stating that such changes would be best done in a separate proposal (clause B.2.6). Most of the changes are based on the algorithmic research done since the standardization of the standard algorithms, and notably on the advances reported in Stepanov and McJones' *Elements of Programming* [3].

This proposal includes the wording for the new versions of the algorithms, as well as updated complexities. It also includes some further comments and benchmarks to demonstrate that weakening the iterator categories of these algorithms is worth it. We propose to make the following algorithms work with forward iterators:

- `std::inplace_merge`

- `std::sort`

- `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`, but not the standard library containers `std::list` and `std::forward_list`, which respectively provide bidirectional and forward iterators. However, both of them have a member function `sort` to perform the job, even though sorting through a member function is hardly a generic approach. Even the recent C++ Core Guidelines [4] 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.

That said, `std::list::sort` and `std::forward_list::sort` have a reason to exist: 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. Moreover, these functions don't invalidate the iterators and even work when the stored elements are not moveable. These improvements are due to the ability that lists have to relink nodes instead of moving values around. 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 that the algorithm is stable).

Basically, we would like to be able to sort pairs forward, bidirectional and random-access iterators through a common algorithm interface. The algorithm `std::inplace_merge` is often a building block of `std::stable_sort` and clearly influences its complexity guarantees; therefore it makes sense to weaken its iterator category to forward iterators too (algorithms exist).

III   Discussion

Complexity of `std::sort`

Some versions of quicksort work reasonably well with forward itera-
tors 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 effi-
cient sorting algorithms such as introsort [5] and pattern-defeating
quicksort [6] only work with random-access iterators because they
need to fall back to a heapsort in order to avoid the quadratic
behavior of quicksort.

We think that the right approach to this problem is to only
guarantee an $\mathcal{O}(n \log n)$ average complexity for forward and bidi-
rectional iterators, and either no worst case complexity or a $\mathcal{O}(n^2)$
worst case complexity. In C++03, `std::sort` has no worst case
complexity guarantee, before efficient algorithms with an $\mathcal{O}(n \log n)$
worst case complexity were discovered. A simple mergesort has a
$\mathcal{O}(n \log n)$ worst case complexity for forward and bidirectional it-
erators, 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 de-
scribed as follows:

> It does at most $N \log^2(N)$ (where N == `last -
> first`) comparisons; if enough extra memory is avail-
> able, 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 itera-
tors, it is easy to make `std::stable_sort` work with bidirectional
iterators as well, with the exact same complexity guarantees as
with 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. In *Elements of Programming* [3], there
is an implementation of an in-place merge algorithm working with
forward iterators which adapts to the available memory. That al-
gorithm runs in $\mathcal{O}(n)$ time hen enough memory is available and in
$\mathcal{O}(n \log n)$ if no additional memory is available, which means that a
mergesort for forward iterators can run in $\mathcal{O}(n \log n)$ or $\mathcal{O}(n \log^2 n)$
time depending on the available memory, which corresponds to the
current complexity of `std::stable_sort` for random-access itera-
tors.

Benefits of `sort` member functions

`std::list::sort` and `std::forward_list::sort` can be more ef-
ficient than general-purpose stable sorting algorithms for forward
and bidirectional iterators; this is due to the fact that they can
relink the list's nodes instead of moving or swapping the values.
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.

While having `std::sort` call `std::list::sort` when given list
iterators would be an interesting idea, the standard generic algo-
rithms are not allowed to alter the containers, and relinking nodes
clearly counts as altering the list; therefore, relinking nodes when
using generic algorithms is not a legal optimization.

To make sure that using a generic version of `std::sort` on lists wouldn't incur a big performance loss, we implemented some basic sorting sorting algorithms for bidirectional and forward iterators and did some benchmarks. While our implementations are not as refined as the ones typically found in standard library implementations, the benchmarks show that some generic sorting algorithms are not orders of magnitude slower than `std::list::sort` and perform rather well. Our sorting algorithms for forward iterators are not that good compared to `std::forward_list::sort` but we believe that good implementations might perform better than ours. Also, note that the benchmarks have been performed with lists of integers and that node relinking might always outperform our sorting algorithms for objects that are expensive to move around.

Parallel sorting algorithms

This proposal focuses on the sequential algorithms from the standard library header `<algorithm>`, but we believe that the parallel algorithms from the Parallelism TS [7] could be other targets for such a change. This paper does not propose to weaken the iterator categories of the corresponding algorithms in the Parallelism Ts, but we encourage people to check whether it can be done and to analyze the resulting algorithmic complexities.

Range algorithms

The Ranges TS [**?**] 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; at most $\mathcal{O}(N \log N)$ if `ForwardIterator` additionally satisfies the requirements of `RandomAccessIterator`.

25.4.1.2 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).

25.4.4 Merge

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

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

8   *Effects:* Merges two sorted consecutive ranges `[first, middle)`
       and `[middle, last)`, putting the result of the merge into
       the range `[first, last)`. The resulting range will be in
       non-decreasing order; that is, for every iterator i in `[first,`
       `last)` other than `first`, the condition `*i < *(i - 1)` or,
       respectively, `comp(*i, *(i - 1))` will be false.

9   *Requires:* The ranges `[first, middle)` and `[middle, last)`
       shall be sorted with respect to `operator<` or `comp`. `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).

10   *Complexity:* When enough additional memory is available,
       `(last - first) - 1` comparisons. If no additional memory

is available, an algorithm with complexity $N \log(N)$ (where N is equal to `last - first`) may be used.

11   *Remarks:* Stable (17.6.5.7).

V   Conclusion

The standard library should make state-of-the-art algorithms available when those ones are easy enough to implement and strive to weaken the iterator categories of these algorithms when doing so occurs no obvious drawback. Sorting algorithms and related algorithms have been studied in depth over the course of the years, and the algorithms discovered since the birth of the standard library allow to efficiently implement some standard algorithms with weakened iterator categories, which is what we propose to standardize with this paper.

Bibliography

[1] B. Stroustrup and A. Sutton. A concept design for the stl. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf

[2] E. Niebler. Working draft, c++ extensions for range. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0021r0.pdf

[3] A. Stepanov and P. McJones, *Elements of Programming*, 1st ed. Addison-Wesley Professional, 2009.

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

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

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

[7] 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