`constexpr` Stable Sorting

Document: P2562R0
Date: March 9, 2022
Project: Programming Language C++, Library Working Group
Audience: LEWG & LWG

Reply to: Oliver J. Rosten (`oliver.rosten@gmail.com`)

Abstract

It is proposed to make `std::stable_sort`, `std::stable_partition`, `std::inplace_merge` and their `ranges` counterparts useable in constant expressions. This applies to overloads which do not accept an execution policy.

CONTENTS

I. INTRODUCTION

C++ 20 saw many of the existing algorithms defined in the `<algorithm>` header declared `constexpr` [P0202]. Those with counterparts in the `ranges` library entered the standard similarly enabled [P0896]. A notable omission, however, is the family of algorithms relating to stable sorting: `std::stable_sort`, `std::stable_partition`, `std::inplace_merge`, and their `ranges` counterparts. The purpose of this paper is to rectify the situation, for those overloads which do not accept an execution policy.

II. MOTIVATION & SCOPE

Since the introduction of `constexpr` in C++11, the past decade has seen ever growing parts of both the core language and library useable in constant expressions. In C++20, there were some significant extensions. Of particular relevance to this paper are that:

1. Many algorithms are now `constexpr`;
2. It is possible to detect if a function call is being used in a `constexpr` context, via `std::is_constant_evaluated()`;
3. Containers may be implemented such that they can be used, to some extent, in a `constexpr` context [P0787]. A concrete, relevant example is `std::vector` [P1004].

The first point suggests that wherever there is the opportunity to bring algorithms into the `constexpr` club, it makes sense to do so: this improves the uniformity of the standard library and removes what may appear to users to be artificial restrictions.

The importance of the second point is subtle and will be discussed in section III. As will be seen, a consequence is that the third point is more motivational than strictly necessary. However, it may provide some interesting opportunities, as will now be discussed. Merge sort is perhaps the canonical example of an efficient stable sort algorithm. It may be straightforwardly implemented using `std::vector` in a way which is now amenable to use in constant expressions.

```
template<class Iter, class OutIter>
constexpr void merge_sort(Iter first,
                          Iter last,
                          OutIter out)
{
  const auto dist{std::distance(first, last)};
  if(dist < 2) return;

  const auto partition{std::next(first, dist / 2)};
  merge_sort(first, partition, out);
  merge_sort(partition, last, std::next(out, dist / 2));
  std::merge(first, partition, partition, last, out);
  std::copy(out, std::next(out, dist), first);
}

template<class Iter>
constexpr void merge_sort(Iter first, Iter last)
{
  using T = typename Iter::value_type;
  std::vector<T> v(first, last);
  merge_sort(first, last, v.begin());
}
```

An example of the code in action can be found on compiler explorer [6]. The key point to note is that it is the availability of extra storage, here provided by `std::vector`, which enables the algorithm to achieve its optimal asymptotic efficiency of $N \ln N$. It is worth noting that a `std::array` cannot be used instead. Tempting code along the lines of std::array<T, std::distance(first, last)> does not compile.

III. STATE OF THE ART

The reality of implementing stable sort (and related algorithms) for the standard library is more subtle than the example above supposes. Most importantly, it is not guaranteed that additional storage is available. Indeed, should this be the case, the standard relaxes the computational complexity requirements. For example, `std::stable_sort`, is allowed to weaken to $N \ln^2 N$ in this situation. Implementations typically deal with this by implementing some sort of merge-without-buffer helper function [7, 8]. Interestingly, these are directly amenable for use in constant expressions, since they can make use of things like iterator arithmetic and `std::rotate`. This begs the question: why aren't the stable sorting algorithms `constexpr` already?

The answer is that implementations branch, dynamically, according to whether or not additional storage is available. For current implementations, the path where it *is* available cannot directly be made useable in constant expressions. This is why the advent of `std::is_constant_evaluated()` is relevant: one solution (though not necessarily the most elegant!) is to ensure that in a `constexpr` context the algorithms of interest statically branch to take a `constexpr`-friendly route. Alternatively, it may be possible to rework implementations roughly along the lines of the `std::vector` example above.

The final subtlety to mention is that while merge sort has unbeatable asymptotic behaviour, it may not be all that fast for small numbers of elements. Therefore, implementations typically resort to cruder sorting methods, such as insertion sort, below some threshold. This does not present any difficulties as far this proposal goes, since there are no barriers to implementing insertion sort in a manner suitable for constant expressions. And even if there were, `std::is_constant_evaluated()` would offer a solution.

IV. IMPACT ON THE STANDARD

This is a pure library extension. Library vendors may either patch existing implementations using `std::is_constant_evaluated()` or rework implementations such that they exploit the extended range of `constexpr` for containers. For the former, an implementation of the essential elements, based on `libstdc++`, can be found on github [9].

REFERENCES

[N4901] Thomas Köppe, ed., Working Draft, Standard for Programming Language C++.

[P0202] Antony Polukhin, Add Constexpr Modifiers to Functions in <algorithm> and <utility> Headers

[P0896] , Eric Niebler, Casey Carter, Christopher Di Bella The One Ranges Proposal

[P0787] , Peter Dimov, Louis Dionne, Nina Ranns, Richard Smith, Daveed Vandevoorde, More constexpr containers

[P1004] , Louis Dionne, Making `std::vector` constexpr

[6] https://godbolt.org/z/n3vEsMr6e

[7] gcc/libstdc++-v3/include/bits/stl_algo.h

[8] libcxx/include/algorithm

[9] https://github.com/ojrosten/sequoia/blob/constexpr_stable_sort/Tests/Experimental/ExperimentalTest.cpp

V. PROPOSED WORDING

The following proposed changes refer to the Working Paper [N4901].

A. Modification to "Header <algorithm> synopsis" [algorithm.syn]

```
// [alg.sorting], sorting and related operations
// [alg.sort], sorting

...

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

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

template<class ExecutionPolicy, class RandomAccessIterator>
   void stable_sort(ExecutionPolicy&& exec,                    // see [algorithms.parallel.overloads]
                    RandomAccessIterator first, RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
   void stable_sort(ExecutionPolicy&& exec,                    // see [algorithms.parallel.overloads]
```

```
                    RandomAccessIterator first, RandomAccessIterator last,
                    Compare comp);

 namespace ranges {
    template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
             class Proj = identity>
      requires sortable<I, Comp, Proj>
      constexpr I stable_sort(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 borrowed_iterator_t<R>
        stable_sort(R&& r, Comp comp = {}, Proj proj = {});
}

...

// [alg.partitions], partitions

...

template<class BidirectionalIterator, class Predicate>
  constexpr BidirectionalIterator stable_partition(BidirectionalIterator first,
                                                   BidirectionalIterator last,
                                                   Predicate pred);

template<class ExecutionPolicy, class BidirectionalIterator, class Predicate>
  BidirectionalIterator stable_partition(ExecutionPolicy&& exec,  // see [algorithms.parallel.overloads]
                                         BidirectionalIterator first,
                                         BidirectionalIterator last,
                                         Predicate pred);

 namespace ranges {
    template<bidirectional_iterator I, sentinel_for<I> S, class Proj = identity,
             indirect_unary_predicate<projected<I, Proj>> Pred>
      requires permutable<I>
      constexpr subrange<I> stable_partition(I first, S last, Pred pred, Proj proj = {});

    template<bidirectional_range R, class Proj = identity,
             indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
      requires permutable<iterator_t<R>>
      constexpr borrowed_subrange_t<R> stable_partition(R&& r, Pred pred, Proj proj = {});
}

...

// [alg.merge], merge

...

template<class BidirectionalIterator>
  constexpr void inplace_merge(BidirectionalIterator first,
                               BidirectionalIterator middle,
                               BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  constexpr void inplace_merge(BidirectionalIterator first,
                               BidirectionalIterator middle,
                               BidirectionalIterator last, Compare comp);

template<class ExecutionPolicy, class BidirectionalIterator>
  void inplace_merge(ExecutionPolicy&& exec,                  // see [algorithms.parallel.overloads]
                     BidirectionalIterator first,
```

```
                     BidirectionalIterator middle,
                     BidirectionalIterator last);

template<class ExecutionPolicy, class BidirectionalIterator, class Compare>
  void inplace_merge(ExecutionPolicy&& exec,                  // see [algorithms.parallel.overloads]
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);

namespace ranges {
    template<bidirectional_iterator I, sentinel_for<I> S, class Comp = ranges::less,
             class Proj = identity>
      requires sortable<I, Comp, Proj>
      constexpr I inplace_merge(I first, I middle, S last, Comp comp = {}, Proj proj = {});

    template<bidirectional_range R, class Comp = ranges::less, class Proj = identity>
      requires sortable<iterator_t<R>, Comp, Proj>
      constexpr borrowed_iterator_t<R>
        inplace_merge(R&& r, iterator_t<R> middle, Comp comp = {}, Proj proj = {});
}
```