Document number:	N1840=05-0100
Document title:	C++0x Proposal: Function template `std::minmax` and / or algorithm `std::minmax_element`
Author:	Hervé Brönnimann
Contact:	hbr@poly.edu
Organization:	Polytechnic University
Date:	2005-06-28
Working Group:	Library

Abstract

This proposal adds a function template (`std::minmax`) and / or an algorithm (`std::minmax_element`) to the header `<algorithm>`, both of which are implicitly present (by two separate calls to `std::min` and `std::max`, or `std::min_element` and `std::max_element`), but admit a better implementation if the two calls are combined into one. Both additions can be considered independently, although they are strongly related hence the common proposal. Both are very low-risk and two reference implementations are given for `std::minmax_element`.

Contents

I Motivation and Scope

This proposal adds a function template (`std::minmax`) and / or an algorithm (`std::minmax_element`) to the header `<algorithm>`. Both can be easily implemented by two separate calls to `std::min` and `std::max`, or `std::min_element` and `std::max_element`, but admit a better implementation if the two calls are combined into one.

The two additions are closely related but independent. It is possible to accept `minmax_element` but reject `minmax`. Of the two, however, `minmax_element` is perceived to be the more important and useful (and I hope the less controversial of the two).

1 The function `minmax` solves the following problem: simultaneous min and max computation requires only one comparison, but using `std::min` and `std::max` forces two comparisons.

2 The purpose of `std::min_element` and `std::max_element` is to determine the *extent* of a range, and it is often the case that both need to be known. The need for knowing the extent of a range is so basic that it hardly needs justification, but here are a few:
1. in order to build a histogram over a range, the extent of the range (both min and max) must be known.
2. in computer graphics, one needs to compute enclosing boxes of a scene, by taking both min and max of the vertex coordinates.

(Note: In the latter example, however, several passes must still be made, one for each coordinate, although the gain per coordinate is the same as in a one-dimensional range. It would be even more efficient to have a single pass algorithm working on all the coordinates at once, e.g. by using a minmax accumulator, but this is beyond the scope of this proposal.)

In the case of `std::minmax_element`, there are two competing implementations. Either of them performs only n `operator++` on the range iterator vs. $2n$ each for the two separate calls to `std::min_element` and `std::max_element`. One is a simple loop combining both `min_element` and `max_element` and performs $2n$ comparisons. The other is an algorithm that performs $3n/2+O(1)$ comparisons. This is a standard exercise in algorithms textbooks [1, Sec 9.1, p. 185], and an adversary argument also provides a lower bound of $\lfloor 3n/2 \rfloor$ comparisons in any case [1, Ex. 9.1-2, p. 185].

This algorithm was incorporated into `www.boost.org` as the Boost minmax library. As documented in the Boost minmax libary webpage, the runtime of `std::minmax_-element` is only slightly higher than `std::min_element` alone, and thus results in a gain of a factor of two compared to the two separate calls (this is especially true for expensive iterators such as `set`).

Some care must be exercised to return the first occurences of min and max elements if several equivalent values occur in the input range. For instance, if all the elements were identical, the algorithm as described in [1] would never return `std::make_-pair(first,first)`. See the reference implementations given below.

3 As an example of usage, with the appropriate `#include`s:

```
int main()
{
  using namespace std;
  // note that this would read better with declaration as: auto result1 = ...;
  pair< reference_wrapper<int const>, reference_wrapper<int const>
    result1 = minmax(1, 0);
  assert( result1.first == 0 );
  assert( result1.second == 1 );

  list<int> L;
  typedef list<int>::const_iterator iterator;
  generate_n(front_inserter(L), 1000, rand);
  pair< iterator, iterator >
    result2 = minmax_element(L.begin(), L.end());
  cout << "The␣smallest␣element␣is␣" << *(result2.first) << endl;
  cout << "The␣largest␣element␣is␣␣" << *(result2.second) << endl;

  assert( result2.first  == std::min_element(L.begin(), L.end());
  assert( result2.second == std::max_element(L.begin(), L.end());
}
```

II Impact On the Standard

This proposal defines two connected but independent pure library extensions.

The impact with `minmax_element` is limited strictly to the addition of the two function templates. It does not interact with other parts of the standard that I can think of.

Although the presence of `minmax` would be preferable for symmetry, it also ties to the acceptance of the reference wrapper (N1453) which is already accepted into the Library TR. The proposal for `minmax` for C++0x is conditional on the presence of reference wrappers in C++0x.

III Design Decisions

1 For `minmax(a,b)`, if the two values `a` and `b` are stored consecutively in a range, one can use `sort`, but it is wasteful and does not work for values stored in non-consecutive positions or in two unrelated variables.

2 Like min and max, minmax returns a pair of references to `T const`. This prevents idioms such as `tie(a,b)=minmax(a,b);` to order two elements `a` and `b`, although this would have the desired effect if we returned a reference instead of a constant reference. The reason is that two unnecessary assignments are performed if `a` and `b` are in order. It is better to stick to `if (b<a) swap(a,b);` to achieve that effect. The proper treatment however involves the presence of the reference wrappers because one cannot instantiate `pair<T const&, T const&>`.

3 The decision to allow up to $2n-2$ comparisons for `minmax_element` is to allow the trivial implementation as well as the more sophisticated one detailed below. In fact, experiments show that the trivial loop performs just as well (in fact, slightly better for vector and set iterators and slightly worse for list iterators), despite making more comparisons. My decision is to leave it up to the implementor of the library.

4 The decision to use a pair of `reference_wrapper<T const>` instead of `tuple<T const&, T const&>` as is done in Boost is motivated by the fact that `<algorithm>` already includes `<utility>` and hence the wrappers, while it seems ridiculous to introduce a dependence on `<tuple>` just for `minmax` (should `<tuple>` be accepted into the standard).

5 The Boost minmax library also provides variants such as `last_min_element`, `first_min_last_max_element`, etc. In this terminology, the standard `std::min_element` corresponds to `first_min_element`, and the proposed `std::minmax_element` corresponds to `first_min_first_max_element`. These variants are somewhat academic and outside the scope of this proposal.

IV Proposed Text for the Standard for `std::minmax`

In the **header <algorithm> synopsis**, add:

```
// 25.3.7, minimum and maximum:
// [ ... min, max, followed by]
    template<class T>
      pair< reference_wrapper<T const>,
            reference_wrapper<T const> >
        minmax(const T& a, const T& b);
    template<class T, class Compare>
      pair< reference_wrapper<T const>,
            reference_wrapper<T const> >
        minmax(const T& a, const T& b, Compare comp);
// [ ... min_element, and max_element]
```

In section 25.3.7, add:

```
    template<class T>
      pair< reference_wrapper<T const>,
            reference_wrapper<T const> >
        minmax(const T& a, const T& b);

    template<class T, class Compare>
      pair< reference_wrapper<T const>,
            reference_wrapper<T const> >
        minmax(const T& a, const T& b, Compare comp);
```

7 **Requires:** Type `T` is `LessThanComparable` (20.1.2) and `CopyConstructible` (20.1.3).

8 **Returns:** `std::make_pair(cref(b),cref(a))` if `b` is smaller than `a`, and `std::make_pair(cref(a),cref(b))` otherwise.

9 **Note:** Returns `(a,b)` if `a` and `b` are equivalent.

10 **Complexity:** Exactly one comparison.

V Proposed Text for the Standard for `std::minmax_element`

In the **header <algorithm> synopsis**, add (and by the way, the comment heading 25.3.8 is missing in the synopsis even in the 2003 revision):

```
// 25.3.7, minimum and maximum:
// [ ... min_element, and max_element, followed by:]
template<class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
    minmax_element (ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
    minmax_element (ForwardIterator first, ForwardIterator last,
                    Compare comp);

// 25.3.8, lexicographical comparisons:
// [ ... lexicographical_compare, etc.]
```

At the end of section 25.3.7, add:

```
template<class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last,
                   Compare comp);
```

11 **Returns:** `std::make_pair(m,M)` where `m` is the `std::min_element` and `M` the `std::max_element` of the input range `[first, last)` for the corresponding comparisons.

12 **Complexity:** At most `max( 2*(last-first)-2, 0)` applications of the corresponding comparisons.

VI Reference implementation of `minmax_element`

The naive implementation of the predicate version of `minmax_element` is simply

```
template <class ForwardIter, class Compare >
pair<ForwardIter,ForwardIter>
minmax_element(ForwardIter first, ForwardIter last,
               Compare comp)
{
  ForwardIter min_result = first, max_result = first;
  if (first != last) {
    while (++first != last) {
      if (comp(*first, *min_result)) min_result = first;
      if (comp(*max_result, *first)) max_result = first;
    }
  }
  return make_pair(min_result, max_result);
}
```

For reference, we present the full code of a reference implementation (see the Boost minmax library) of the binary predicate version of the algorithm that performs an optimal number of comparisons. It is a lot more complex and in practice performs about the same (sometimes slightly more, sometimes slightly less) despite the lower number of comparisons. On the other hand, it is not much worse either and guarantees an optimal number of comparisons.

```
template <class ForwardIter, class Compare >
pair<ForwardIter,ForwardIter>
minmax_element(ForwardIter first, ForwardIter last, Compare comp)
{
  // 1. if no elements
  if (first == last)
    return make_pair(last,last);

  // 2. declare return values
  ForwardIter min_result = first;
  ForwardIter max_result = first;

  // 3. if only one element
  ForwardIter second = first; ++second;
  if (second == last)
    return make_pair(min_result, max_result);

  // 4. treat first pair separately (only one comparison for first two elements)
  ForwardIter potential_min_result = last;
  if (comp(*first, *second))
    max_result = second;
  else {
    min_result = second;
    potential_min_result = first;
  }
  // 5. then each element by pairs, with at most 3 comparisons per pair
  first = ++second; if (first != last) ++second;
  while (second != last) {
    if (comp(*first, *second)) {
      if (comp(*first, *min_result)) {
        min_result = first;
        potential_min_result = last;
      }
      if (comp(*max_result, *second))
        max_result = second;
    } else {
      if (comp(*second, *min_result)) {
        min_result = second;
        potential_min_result = first;
      }
      if (comp(*max_result, *first))
        max_result = first;
    }
    first = ++second; if (first != last) ++second;
  }

  // 6. if odd number of elements, treat last element
  if (first != last) { // odd number of elements
    if (comp(*first, *min_result)) {
      min_result = first;
      potential_min_result = last;
    } else if (comp(*max_result, *first))
      max_result = first;
  }

  // 7. resolve min_result being incorrect with one extra comparison
  // (in which case potential_min_result is necessarily the correct result)
  if (potential_min_result != last
      && !comp(*min_result, *potential_min_result))
    min_result = potential_min_result;

  return make_pair(min_result,max_result);
}
```

Note the following points:

- Exit is performed as early as possible in sections 1 or 3 with no comparisons, which is the same as the naive implementation.
- In case of two elements, we perform only two comparisons (one in section 4 and another in section 7), which is the same as the naive implementation.
- For more than 2 elements, the first pair only induces one comparison, but every subsequent pair induces 3 comparisons, the last element induces two comparisons if it is at an even position (section 6), and there is perhaps one more comparison at the end (section 7), thus the number of comparisons is at least

$3\lfloor n/2 \rfloor - 2$ if n is even and $3\lfloor n/2 \rfloor$ if n is odd, and at most $3\lfloor n/2 \rfloor - 1$ if n is even and $3\lfloor n/2 \rfloor + 1$ if n is odd.

- The bulk of the algorithm is in section 5, where 3 comparisons per consecutive pair of iterators are performed. We are careful to only perform the minimum required number of `operator++`.
- The only subtle point of the implementation is that the loop as given in section 5 does not quite keep the first occurence of the minimum element, in the case when `*first` and `*second` are equivalent and trigger an update of the minimum, because in this case the `else` clause is evaluated, we compare `*min_result` to `*second` and it is `second` that updates `min_result`. So we must make a comparison at the end to resolve this case (section 7).

This extra comparison results from the desire to return the *first* occurence of both minimum and maximum elements, which is not possible if elements are treated in pair and either the result can only be `std::make_pair(first,second)` or `std::make_pair(second,first)`.

VII Acknowledgements

My students in CS903 (Polytechnic Univ., http://photon.poly.edu/ hbr/cs903/) who had minmax_element as an assignment helped clarify the issues, and also came up with the optimum number of comparisons for first_min_last_max_element.

One minmax_element implementation, which performs 3(n/2)+O(log n) comparisons on the average when the elements are random_shuffled, was suggested by my student Marc Glisse. The current one, which performs 3(n/2)+1 comparisons in the worst case, was suggested by John Iacono.

Finally, Matthew Wilson and Jeremy Siek contributed pre-review comments, while Gennadiy Rozental, John Maddock, Craig Henderson, Gary Powell participated in the review of the Boost library, managed by Thomas Witt. Late after the review, as I finally scrounged to add the library for a release, Eric Niebler noted the bad behavior of pair for minmax and suggested to use Boost.tuple instead. Here I use reference wrappers instead. All my thanks for the excellent advice and reviews from all.

References

[1] T. Cormen, C. Leiserson, R. Rivest and C. Stein. *Introduction to algorithms (2nd edition)*. The MIT Press, 2004.