Doc No: N2551=08-0061

A variadic std::min(T, ...) for the C++ Standard Library (Revision 2)

Sylvain Pion*

2008-02-28

Abstract

We propose a small handy extension to the functions `std::min`, `std::max` and `std::minmax`, so that they can handle more than 2 arguments. This is a low hanging fruit allowed by the variadic template feature.

I History of changes to this document

Since N2485, the following has been changed:

- Made the proposal a pure addition by keeping the current signatures
- Added a variadic version of the overloads taking a comparison functor
- Removed dependency on concepts
- Added complete wording
- Added sample implementation

II Motivation and Scope

Let us consider the following use case.

It is sometimes needed to take the minimum of 3 or more values. Doing it with the C++98 standard library requires something like:

```
m = std::min(a, std::min(b, c));
m = std::min(a, std::min(b, std::min(c, d)));
m = std::min(a, std::min(b, std::min(c, d, cmp), cmp), cmp);
```

The proposal would allow to simply write:

```
m = std::min(a, b, c);
m = std::min(a, b, c, d);
m = std::min(a, b, c, d, cmp);
```

In the sequel, we will only mention `std::min` for simplicity, but we always mean it for the 3 functions `std::min`, `std::max` and `std::minmax`.

*INRIA Sophia-Antipolis, France. Sylvain.Pion@sophia.inria.fr

III Impact on the Standard

The only incompatibility introduced is the case where the C++98 `std::min` function is called with a third argument (the comparison functor), in the case where the functor has the same type as the arguments. With this proposal, this will now call the variadic `std::min` instead. We think that this is very unlikely to have ever been encountered in practice. In the case where this would be considered a problem, we would propose a different name for the new functionality, like `min_n`.

Concepts would help to make this proposal easier to specify and implement: by constraining all arguments to be of the same type with a `requires SameType<T, Args>...` close. Without concepts, `enable_if` works just as well.

IV Design Decisions

The one argument special case:

Although this is useless in isolation, the one argument special case makes it uniform, and is meant to be used in a variadic template context where the number of arguments could happen to be one. Moreover, the constraints on the one argument overload are relaxed, as the type does not need to be `LessThanComparable` in this case (this issue does not appear in the non-concept version). Note that a zero argument version would not make sense, as a return type is needed.

The comparison functor:

For homogeneity reasons, it is desirable to similarly extend the `std::min` overload taking a comparison functor. The natural extension is to keep the comparison functor as the last argument of the function call, this way:

```
template < typename T,
           BinaryPredicate<T, T> Compare,
           typename... Args >
requires SameType<T, Args>...
const T&
min(const T& a, const T& b, const Args&... args, Compare comp);
```

Recall that variadic template arguments packs can only appear at the end of the argument list of a function, so this is not valid syntax. It is still possible to define an equivalent function, by tweaking the constraints on the list of types of `Args`, so that the last one should match the `Compare` type instead of being `T`. One minor knit with this, unfortunately, is that it is still not possible to pass this last argument by value while all others are passed by const reference.

An alternative would be to move the comparison functor as the first argument, and eventually providing a different name for the function. This would of course break the homogeneity with the current 3 argument `std::min` function.

Complexity:

We chose to use the terms of LWG issue 715.

Choice in case of equivalence:

For `minmax`, the maximum value is chosen to be the last one found in case there are several equivalent. This is in line with the current specification of `minmax`, and the one provided by issue 715 for `minmax_element`.

Making it only an addition

A question was raised on lowering the risk for cases like calling with explicit template parameters: `std::min<T,Compare>(a,b,cmp)`, therefore keeping the current signatures, and only adding new ones to handle the variadic case. This in turn would trigger the need for adding signatures for `std::min(T, T, T)`.

V Proposed wording

Based on the working draft N2521:

In 25.0.2:

In the block:

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

template<class T> const T& max(const T& a, const T& b);
template<class T, class Compare>
  const T& max(const T& a, const T& b, Compare comp);

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

Add the following (keeping `min` overloads together, etc.):

```
template<class T> const T& min(const T& a, const T&b, const T&c);
template<class T, class... Args> const T& min(const T& a, const Args&... args);
template<class T, class U, class... Args>
  const T& min(const T& a, const U& b, const Args&... args);

template<class T> const T& max(const T& a, const T&b, const T&c);
template<class T, class... Args> const T& max(const T& a, const Args&... args);
template<class T, class U, class... Args>
  const T& max(const T& a, const U& b, const Args&... args);

template<class T> pair<const T&, constT&> minmax(const T& a, const T&b, const T&c);
template<class T, class... Args>
  pair<const T&, const T&> minmax(const T& a, const Args&... args);
template<class T, class U, class... Args>
  pair<const T&, const T&> minmax(const T& a, const U& b, const Args&... args);
```

In 25.3.7:

In the block:

```
  template<class T> const T& min(const T& a, const T& b);
  template<class T, class Compare>
    const T& min(const T& a, const T& b, Compare comp);
Requires: Type T is LessThanComparable.
Returns: The smaller value.
Remarks: Returns the first argument when the arguments are equivalent.

  template<class T> const T& max(const T& a, const T& b);
  template<class T, class Compare>
    const T& max(const T& a, const T& b, Compare comp);
Requires: Type T is LessThanComparable.
Returns: The larger value.
Remarks: Returns the first argument when the arguments are equivalent.

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

```
  template<class T, class Compare>
    pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
Requires: Type T shall be LessThanComparable.
Returns: pair<const T&, const T&>(b, a) if b is smaller than a, and
           pair<const T&, const T&>(a, b) otherwise.
Remarks: Returns pair<const T&, const T&>(a, b) when the arguments are equivalent.
Complexity: Exactly one comparison.
```

Add the following (keeping `min` overloads together, etc.):

```
  template<class T> const T& min(const T& a, const T&b, const T&c);
  template<class T, class... Args> const T& min(const T& a, const Args&... args);
Requires: Type T is LessThanComparable, and all types forming Args... are the same as T.
Returns: The smallest value in the set of all the arguments.
Remarks: Returns the leftmost argument when several arguments are equivalent
         to the smallest.  Returns a if sizeof...(Args) is 0.

  template<class T, class U, class... Args>
    const T& min(const T& a, const U& b, const Args&... args);
Requires: The types of all arguments except the last one are the same as T.
          The last argument is a binary predicate over T.
Returns: The first element in a partial ordering of all the arguments except the last one,
         where the ordering is defined by the predicate.
Remarks: Returns the leftmost argument when several arguments are equivalent
         to the first element in the ordering.  Returns a if sizeof...(Args) is 0.

  template<class T> const T& max(const T& a, const T&b, const T&c);
  template<class T, class... Args> const T& max(const T& a, const Args&... args);
Requires: Type T is LessThanComparable, and all types forming Args... are the same as T.
Returns: The largest value in the set of all the arguments.
Remarks: Returns the leftmost argument when several arguments are equivalent
         to the largest.  Returns a if sizeof...(Args) is 0.

  template<class T, class U, class... Args>
    const T& max(const T& a, const U& b, const Args&... args);
Requires: The types of all arguments except the last one are the same as T.
          The last argument is a binary predicate over T.
Returns: The last element in a partial ordering of all the arguments except the last one,
         where the ordering is defined by the predicate.
Remarks: Returns the leftmost argument when several arguments are equivalent
         to the last element in the ordering.  Returns a if sizeof...(Args) is 0.

  template<class T>
    pair<const T&, const T&> minmax(const T& a, const T&b, const T&c);
  template<class T, class... Args>
    pair<const T&, const T&> minmax(const T& a, const Args&... args);
Requires: Type T is LessThanComparable, and all types forming Args... are the same as T.
Returns: pair<const T&, const T&>(x, y) where x is the first element and y the last
         element in a partial ordering of all the arguments.
Remarks: x is the leftmost argument when several arguments are equivalent
         to the smallest.  y is the rightmost argument when several arguments
         are equivalent to the largest.  Returns pair<const T&, const T&>(a, a)
         if sizeof...(Args) is 0.
Complexity: At most (3/2) sizeof...(Args) applications of the corresponding predicate.

  template<class T, class U, class... Args>
    pair<const T&, const T&> minmax(const T& a, const U& b, const Args&... args);
Requires: The types of all arguments except the last one are the same as T.
          The last argument is a binary predicate over T.
```

```
Returns: pair<const T&, const T&>(x, y) where x is the first element and y the last
         element in a partial ordering of all the arguments defined by the predicate.
Remarks: x is the leftmost argument when several arguments would order equivalent
         as first in the ordering.  y is the rightmost argument when several
         arguments would order equivalent as last in the ordering.
         Returns pair<const T&, const T&>(a, a) if sizeof...(Args) is 0.
Complexity: At most (3/2) sizeof...(Args) applications of the corresponding predicate.
```

VI Implementation

Here is an implementation for the proposed `std::min` and `std::minmax` (in namespace `cxx` here):

```
#include <cassert>
#include <algorithm>
#include <type_traits>
#include <functional>
#include <utility>

using std::pair;

namespace cxx {
    namespace detail {

        // Helper that extracts the last type of a pack
        template <class... Args>
        struct last;

        template <class T>
        struct last<T> { typedef T type; };

        template <class T, class... Args>
        struct last<T, Args...> : last<Args...> {};

        // Helper that extracts the last argument of a pack
        template <class T>
        const T&
        last_arg(const T& t)
        { return t; }

        template <class T, class... Args>
        typename last<Args...>::type const &
        last_arg(const T&, const Args&... args)
        { return last_arg(args...); }


        // Internal min() for the non-functor version.
        template <class T>
        const T&
        min(const T&a)
        { return a; }

        template <class T, class... Args>
        const T&
        min(const T&a, const T&b, const Args&... args)
        { return min(b < a ? b : a, args...); }


        // Internal minc(T, ..., cmp) function.
        template <class T, class Cmp>
        const T&
        minc(const T&a, Cmp)
        { return a; }

        template <class T, class... Args>
        const T&
        minc(const T&a, const T&b, const Args&... args)
```

```
        { return minc(last_arg(args...)(b, a) ? b : a, args...); }


        // Internal minmax
        template <class T>
        pair<const T&, const T&>
        minmax(const T&a)
        { return pair<const T&, const T&>(a, a); }

        template <class T>
        pair<const T&, const T&>
        minmax(const T&a, const T&b)
        { return b < a ? pair<const T&, const T&>(b, a)
                       : pair<const T&, const T&>(a, b); }

        // Internal minmaxp() is an intermediate function processing elements 2 by 2
        // to achieve the required complexity of 3N/2 comparisons.
        template <class T>
        pair<const T&, const T&>
        minmaxp(pair<const T&, const T&> cur)
        { return cur; }

        template <class T>
        pair<const T&, const T&>
        minmaxp(pair<const T&, const T&> cur, const T&c)
        {
            if (c < cur.first)
                return pair<const T&, const T&>(c, cur.second);
            if (c < cur.second)
                return cur;
            return pair<const T&, const T&>(cur.first, c);
        }

        template <class T, class... Args>
        pair<const T&, const T&>
        minmaxp(pair<const T&, const T&> cur, const T&c, const T&d, const Args&... args)
        {
            pair<const T&, const T&> r = minmax(c, d);
            pair<const T&, const T&> merged (r.first < cur.first
? r.first : cur.first,
                                             r.second < cur.second ? cur.second : r.second);
            return minmaxp(merged, args...);
        }

        template <class T, class... Args>
        pair<const T&, const T&>
        minmax(const T&a, const T&b, const T&c, const Args&... args)
        { return minmaxp(minmax(a, b), c, args...); }


        // Internal minmaxc(T, ..., cmp) function.
        template <class T, class Cmp>
        pair<const T&, const T&>
        minmaxc(const T&a, Cmp)
        { return pair<const T&, const T&>(a, a); }

        template <class T, class Cmp>
        pair<const T&, const T&>
```

```
        minmaxc(const T&a, const T&b, Cmp cmp)
        { return cmp(b, a) ? pair<const T&, const T&>(b, a)
                           : pair<const T&, const T&>(a, b); }

        // minmaxcp() is an intermediate function processing elements 2 by 2
        // to achieve the required complexity of 3N/2 comparisons.
        template <class T, class Cmp>
        pair<const T&, const T&>
        minmaxcp(pair<const T&, const T&> cur, Cmp)
        { return cur; }

        template <class T, class Cmp>
        pair<const T&, const T&>
        minmaxcp(pair<const T&, const T&> cur, const T&c, Cmp cmp)
        {
            if (cmp(c, cur.first))
                return pair<const T&, const T&>(c, cur.second);
            if (cmp(c, cur.second))
                return cur;
            return pair<const T&, const T&>(cur.first, c);
        }

        template <class T, class U, class... Args>
        pair<const T&, const T&>
        minmaxcp(pair<const T&, const T&> cur,
                 const T&c, const T&d, const U&e, const Args&... args)
        {
            pair<const T&, const T&> r = minmaxc(c, d, last_arg(e, args...));
            pair<const T&, const T&> merged
                 (last_arg(e, args...)(r.first,  cur.first)  ? r.first
: cur.first,
                  last_arg(e, args...)(r.second, cur.second) ? cur.second : r.second);
            return minmaxcp(merged, e, args...);
        }

        template <class T, class... Args>
        pair<const T&, const T&>
        minmaxc(const T&a, const T&b, const T&c, const Args&... args)
        { return minmaxcp(minmaxc(a, b, last_arg(c, args...)), c, args...); }


        // Tool : should std::is_same<> be generalized to variadic number of args?
        template <class T, class... Args>
        struct are_same;

        template <class T>
        struct are_same<T> : std::is_same<T, T> {};

        template <class T, class U, class... Args>
        struct are_same<T, U, Args...> {
            enum { value = (std::is_same<T, U>::value && are_same<T, Args...>::value) };
        };


        // Tool : for the minmax() with comparator.
        template <class T, class U, class... Args>
        struct are_same_except_last;
```

```
    template <class T, class U>
    struct are_same_except_last<T, U> {
        enum { value = ! std::is_same<T, U>::value };
    };

    template <class T, class U, class V, class... Args>
    struct are_same_except_last<T, U, V, Args...> {
        enum { value = (std::is_same<T, U>::value &&
                        are_same_except_last<T, V, Args...>::value) };
    };

} // detail

// Existing functions kept:
template < class T >
const T& min(const T&a, const T&b) { return std::min(a, b); }

template < class T, class Compare >
const T& min(const T&a, const T&b, Compare cmp) { return std::min(a, b, cmp); }

template < class T >
pair<const T&, const T&>
minmax(const T&a, const T&b) { return std::minmax(a, b); }

template < class T, class Compare >
pair<const T&, const T&>
minmax(const T&a, const T&b, Compare cmp) { return std::minmax(a, b, cmp); }

// Additions proposed:
template <class T, class... Args>
typename std::enable_if<detail::are_same<T, Args...>::value,
                        const T&>::type
min(const T&a, const Args&... args)
{ return detail::min(a, args...); }

template <class T, class U, class... Args>
typename std::enable_if<detail::are_same_except_last<T, U, Args...>::value,
                        const T&>::type
min(const T&a, const U&b, const Args&... args)
{ return detail::minc(a, b, args...); }

template < class T >
const T&
min(const T&a, const T&b, const T&c)
{ return min(a, min(b, c)); }


template <class T, class... Args>
typename std::enable_if<detail::are_same<T, Args...>::value,
                        pair<const T&, const T&>>::type
minmax(const T&a, const Args&... args)
{ return detail::minmax(a, args...); }

template <class T, class U, class... Args>
typename std::enable_if<detail::are_same_except_last<T, U, Args...>::value,
                        pair<const T&, const T&>>::type
minmax(const T&a, const U&b, const Args&... args)
{ return detail::minmaxc(a, b, args...); }
```

```
    template < class T >
    pair<const T&, const T&>
    minmax(const T&a, const T&b, const T&c)
    { return minmax(a, b, b, c); }

} // cxx


// Tests:
struct A {};
struct CmpA { bool operator()(const A&, const A&) const { return true; } };

template < class T >
bool equals(const pair<const T&, const T&> &a, const T& b, const T& c)
{
    return a.first == b && a.second == c;
}

// test derived types (currently works when qualifying the template arguments)
struct base {};
struct derived : base {};
bool operator<(const base&, const base&) { return true; }

// Special class which is a functor for comparing itself...
struct special {
        int i;
        special(int ii) : i(ii) {}

        // tests in reverse order compared to std::less.
        bool operator()(const special& a, const special&b) const
        {
                return a.i > b.i;
        }
};

bool operator<(const special& a, const special& b)
{ return a.i < b.i; }


int main()
{
    typedef pair<const A&, const A&> P_A;

    A a;
    CmpA cmp;

    // min()

    // C++98:
    assert( 1 == cxx::min(1, 2) );
    assert( 1.0 == cxx::min(1.0, 2.0) );
    a = cxx::min(a, a, cmp);

    // New:
    assert( 1 == cxx::min(1) );
    assert( 1 == cxx::min(1, 2, 3) );
    assert( 1 == cxx::min(1, 2, 3, 4) );
```

```
    assert( 1.0 == cxx::min(1.0) );
    assert( 1.0 == cxx::min(1.0, 2.0, 3.0, 4.0) );

    a = cxx::min(a, cmp);
    a = cxx::min(a, a, cmp);
    a = cxx::min(a, a, a, cmp);
    a = cxx::min(a, a, a, a, cmp);

    base b;
    derived d;
    const base & m = cxx::min<base>(d, b);
    const base & m2 = cxx::min<base, std::less<base>>(d, b, std::less<base>());

    // Test for changed behavior in the 3 arguments case
    // with T the same type as the functor.
    special s1(1), s2(2), s3(3);
    assert((1 == cxx::min(s1, s2, s3).i));
    assert((2 == std::min(s1, s2, s3).i));


    // minmax()

    // N2521:
    assert(( equals(cxx::minmax(1, 2), 1, 2) ));
    assert(( equals(cxx::minmax(1.0, 2.0), 1.0, 2.0) ));
    P_A p = cxx::minmax(a, a, cmp);

    // New:
    assert(( equals(cxx::minmax(1), 1, 1) ));
    assert(( equals(cxx::minmax(1, 2, 3), 1, 3) ));
    assert(( equals(cxx::minmax(1, 2, 3, 4), 1, 4) ));
    assert(( equals(cxx::minmax(1.0), 1.0, 1.0) ));
    assert(( equals(cxx::minmax(1.0, 2.0, 3.0, 4.0), 1.0, 4.0) ));
    assert(( equals(cxx::minmax(2.0, 1.0, 4.0, 3.0), 1.0, 4.0) ));

    P_A p1 = cxx::minmax(a, cmp);
    P_A p2 = cxx::minmax(a, a, cmp);
    P_A p3 = cxx::minmax(a, a, a, cmp);
    P_A p4 = cxx::minmax(a, a, a, a, cmp);
}
```