Doc No: N2485=07-0355

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

Sylvain Pion[*]

2007-12-07

**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 and concept features.

## I 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)));
```

The proposal would allow to simply write:

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

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`. We will discuss the overloads taking an additional comparison functor specifically.

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

## II Impact on the Standard

It is a generalization of the signature of a function in the standard library.

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.

Variadic templates obviously help defining the function, although it is possible to define overloads up to a fixed number of arguments (e.g. using the preprocessor), if an implementation does not (yet) support variadic templates.

Concepts also make it easier to propose, as they complement the variadic template, by constraining all arguments to be of the same type. Again, in case of lack of support for concepts in an implementation, it is possible to use SFINAE for doing this.

## III Design Decisions

The proposal simply consists in replacing the function `std::min` taking 2 arguments, by the following 2 functions (with a possible one-line implementation):

```
template <typename T>
const T&
min(const T& a)
{ return a; }

template < LessThanComparable T, typename... Args>
requires SameType<T, Args>...
const T&
min(const T& a, const T& b, const Args&... args)
{ return std::min( b < a ? b : a, args...); }
```

Let us now discuss some design choices.

**The one argument special case:**

Although this may seem 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 at instantiation time. Moreover, the constraints on the one argument overload are relaxed, as the type does not need to be `LessThanComparable` in this case. Note that a zero argument version would not make sense, as a type is needed at least for uniformity in the return type.

**The comparison functor:**

For homogeneity reasons, it would be desirable to similarly extend the `std::min` overload taking a comparison functor. We finally opted for **not proposing it**, due to the constraints explained below.

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 would be possible to define a function emulating this, 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`. This would be ugly, though. Moreover, unfortunately, it would still not be 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.

Moreover, we believe that this variant of `std::min` is less useful than the one which hardcodes `operator<`, and that it will become even less useful with concepts, as it will be possible to specify a concept map for LessThanComparable to replace the comparison functor (in some cases). Therefore, extending it seems less important.

This question could be reconsidered if the variadic template feature was extended, such that it is allowed to appear not necessarily at the end of the argument list. But such a change is currently not proposed, and it is not clear if there will ever be enough motivation for such a general change. Nevertheless, this is an additional reason for not proposing a workaround solution now.

**Wording:**

We suggest to use a specification similar to the one used for `std::min_element`. This could lead to something like:

```
Returns: The first argument i such that for any argument j the following
condition holds: !(j < i).

Complexity: Exactly sizeof...(Args)+1 comparisons.
```

**Completeness:**

We do not see any other places in the standard library which could benefit from a similar treatment, so far. One remark could be to provide an extension of the `SameType` concept so that it directly takes a variadic number of arguments. That is, it would replace `SameType<T, Args>...` by `SameType<T, Args...>`. This remark is only food for thought.

## IV Acknowledgements

We would like to thank Doug Gregor for his answers and suggestions.

Moreover, we would like to thank the implementors of the variadic template and concept features in GCC, which (partially so far) allowed for testing.