Variadic functions: Variadic templates or initializer lists? -- Revision 1

Author: Loïc Joly (loic.actarus.joly@numericable.fr)
Document number: **N2772=08-0282**
Date: 2008-09-17
Project: Programming Language C++, Library Working Group
Revised by: Robert Klarer (klarer@ca.ibm.com)

Revision History

Implementation experience revealed that the code changes recommended in the original paper didn't quite work. Minor corrections have been made to the "Recommendations" section at the end of the paper.
– Robert.

Introduction

There are two possibilities to specify a function that take an indefinite number of elements, all of the same type, and both are used at some point in the SL.

Variadic templates

The first way to define it is by variadic template, potentially specifying through concepts that all types should be equal. This method is for instance used for the new variadic min function. It looks a little bit like:

```
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 );
```

The use of this function is the following:

```
int i = min(1, 2, 42, a, b, 34, comp);
```

Note: the real definition is much more complex because the comparator argument is part of the variadic argument pack and need special processing.

Initializer_list

With initializer_list and the new initialization syntax, the same function could be declared this way:

```
template< typename T, BinaryPredicate <T, T> Compare>
const T& min (initializer_list<T> values, Compare comp);
```

The use of this function would be the following:

```
int i = min({1, 2, 42, a, b, 34}, comp);
```

This syntax is actually used in many places in the SL, mostly to add elements to containers.

Possible solutions

Having two solutions for one problem means a choice has to be made. The goal of this paper is to try and establish guidelines, and potentially propose according changes to the SL.

Three solutions are possible:

- Use variadic template
- Use initializer_list
- Provide two overloads: One with initializer_list, one with variadic template

Syntactic considerations

User side

In the case where all parameters are the "variadic" ones (like the min function with no user specified comparator), the initializer_list versions requires typing two more characters ('{' and '}').

On the other hand, those {} have a value of packing together what is logically together. This becomes especially blatant for the version of min that uses a user defined comparator.

The variadic template syntax that appears in the function declaration, and most of the time in the associated concept syntax, are rather obscure. It has been stated by several people with teaching experience that they do not wish to present their students with this syntax until they reach an advanced course, whereas the initializer_list syntax seems easier to teach.

The number of elements of the variadic template version can be known at compile time, while an initializer_list does not seem to be able to do so. For this very reason, for instance, a variadic template make function has been added to std::array.

Implementer side

The variadic template version is harder to write. It also require a template, which may make it unsuitable in some cases (virtual functions, dynamic libraries).

Performance considerations

Theory

The variadic template version creates several instantiations of the function, which might create some code bloat if the function is called with various number of elements.

If the arguments of the function are literals, the performance is basically the same for both solutions. It might even lean in favor of initializer_list if such a list can be a constexpr.

If the arguments can only be be computed at runtime, then for initializer_list, they have to be copied at runtime into an array to which the initializer_list will point. Then the initializer_list itself (two words) will have to be pushed on the stack. In the variadic template solution, values will have to be constructed on the stack. In case the function is inlined, there should be no argument passing cost, whereas the array construction stuff will probably remain. This can lead to some inefficiency for initializer_list.

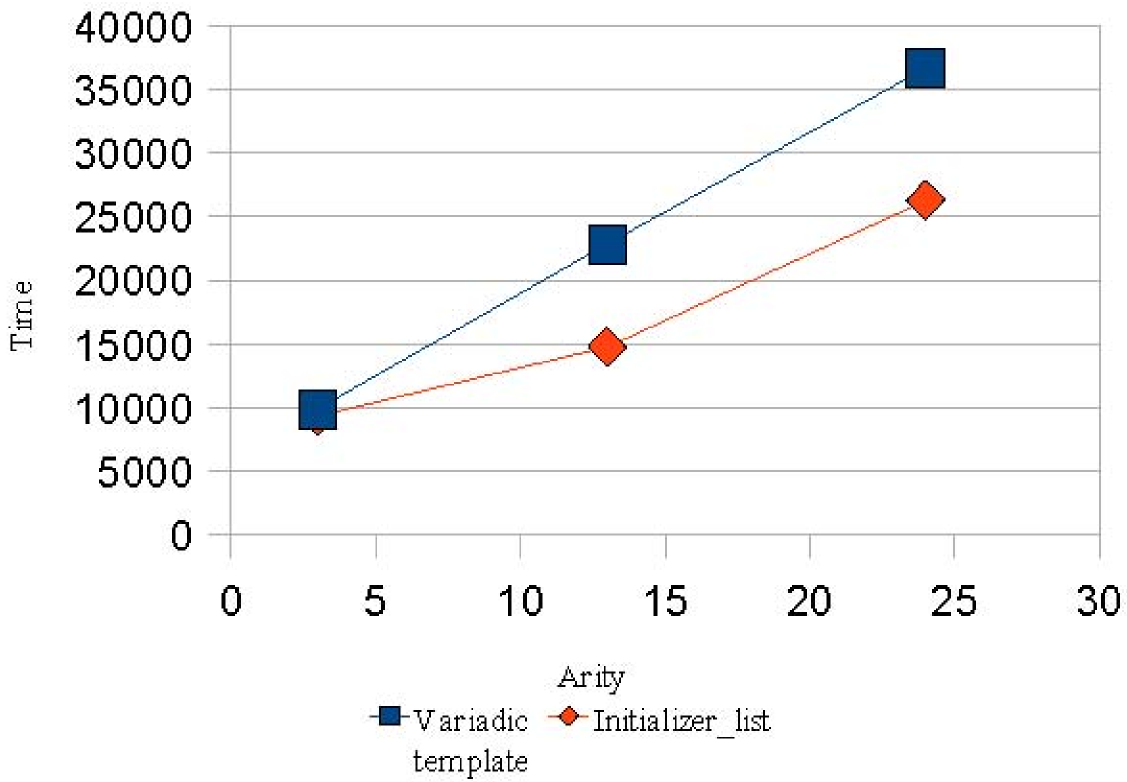
The implementation of an initializer_list version will probably use a loop over the list, whereas a variadic template version is unrolled at compilation time.

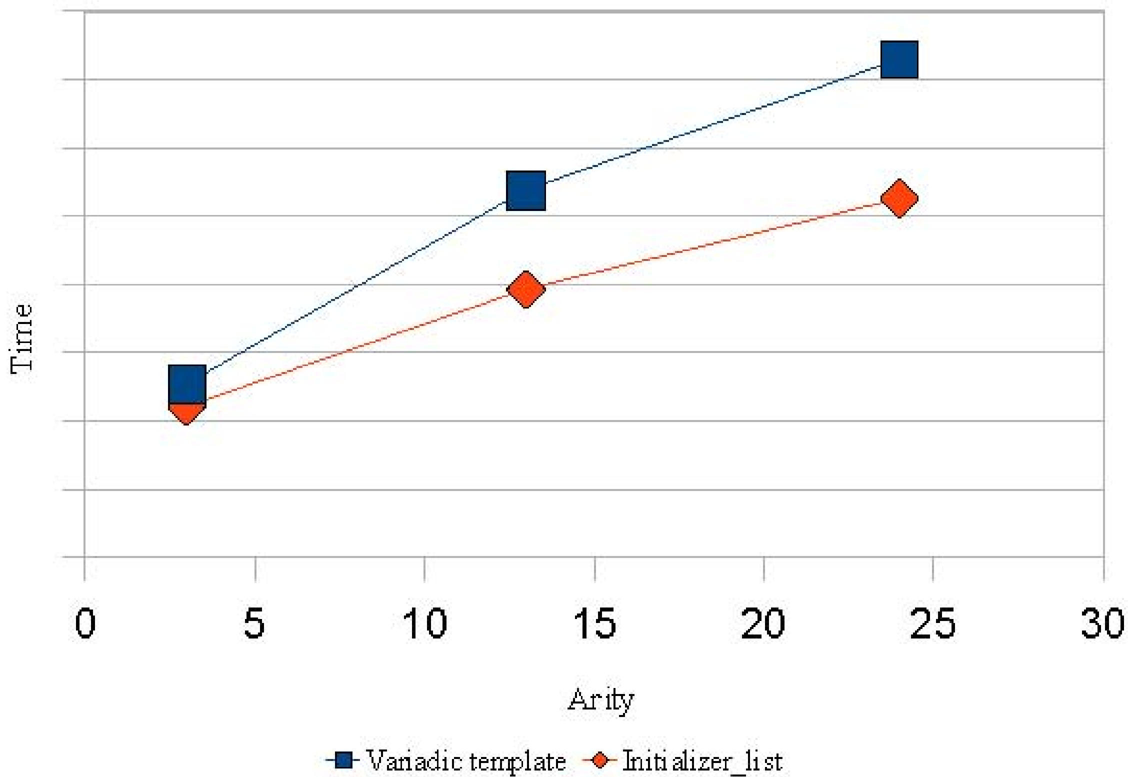
The data inside the initializer_list cannot be moved from, which may introduce some cost for variadic sink.

Benchmark

I made some test with an experimental version of a compiler (gcc) that provided both variadic templates and initializer_lists. The test code is available on request.

I computed the min of 3, 13 or 24 elements with the default comparison operator, the element being either an int or a std::string (30 char long). Here are the results





We can see that in both cases, and perhaps surprisingly considering the theoretical performance considerations, the initializer_list version consistently outperformed the variadic template version.

Recommendations

The initializer_list option provide a clearer syntax than variadic templates, and (unless further evidence contradicts this benchmark) better performances. It should therefore be the preferred choice in the SL. One exception is when the number of elements has to be known at compile time, which an initializer_list cannot provide in the current standard.

In the current draft of the standard, only the minimum and maximum functions do not follow those guidelines. Here are the proposed changes:

In 25.3.7 Minimum and maximum

For **std::min**

Replace (§4, 5 and 6):

```
template<class T, class... Args>
const T& min(const T& a, const Args&... args);
Requires: 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.
```

With

```
template<class T>
T min(initializer_list<T> t);
Requires: T is LessThanComparable and CopyConstructible.
Returns: The smallest value in the initializer_list.
Remarks: Returns the leftmost argument when several arguments are equivalent
to the smallest.
```

And (§7, 8 and 9)

```
template<class T, class U, class... Args>
const T& min(const T& a, const U& b, const Args&... args);
Requires: The types of all the 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.
```

With

```
template<class T, class Compare>
T min(initializer_list<T> t, Compare comp);
Requires: Type T is LessThanComparable and CopyConstructible.
Returns: The smallest value in the initializer_list.
Remarks: Returns the first argument when the several arguments are equivalent
to the smallest.
```

The same modifications should be applied to **std::max** and **std::minmax**.

I would like to thank Alisdair Meredith and Sylvain Pion who provided helpful comments on an early version of this document.