Extending Variadic Template Template Parameters

Authors: Douglas Gregor, Indiana University
Eric Niebler, Boost Consulting
Document number: N2488=07-0358
Date: 2007-12-10
Project: Programming Language C++, Evolution Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

1 Introduction

Variadic templates were explicitly designed to eliminate excessive code repetition in modern template libraries, by providing the ability to express a template that accepts an arbitrary number of parameters in a type-safe manner [2, 1]. Further experimentation with variadic templates in other real-world template libraries has uncovered a limitation in the present form of variadic templates, due to overly strict matching of template template parameters. Although the problem was found within the context of Eric Niebler's Boost.Proto library for building domain-specific embedded languages, this proposal outlines the problem in the context of the Boost Metaprogramming Library [3] and provides proposed wording for changes that address this problem. We have implemented these changes in the GNU C++ front end and verified that they eliminate the need for code repetition in MPL and Boost.Proto.

2 MPL Lambda Expressions with Variadic Templates

The Boost Metaprogramming Library (MPL) makes use of compile-time lambda expressions for performing type computations. For instance, the following code transforms a list of types into a list of pointers to those types.

```
typedef mpl::vector<int, short, float> v;
typedef mpl::transform<v, add_pointer<_1> >::type v2;

BOOST_MPL_ASSERT((is_same<mpl::at_c<v2, 0>::type, int*>));
BOOST_MPL_ASSERT((is_same<mpl::at_c<v2, 1>::type, short*>));
BOOST_MPL_ASSERT((is_same<mpl::at_c<v2, 2>::type, float*>));
```

The expression add_pointer<_1> is an MPL lambda expression, and _1 is a placeholder. The transform line says, "for each type T in the vector, replace _1 with T and evaluate add_pointer<T>". Within the MPL, a so-called meta-function like add_pointer<T> is evaluated by looking for a nested ::type typedef.

This is implemented in the MPL by ripping apart template types, replacing placeholders, and reassembling the types. The key enabling technology for this is template template parameters. There is some template in MPL for evaluating lambdas, such as:

```
template<typename T>
struct eval;
```

Then there are a series of specializations, such as:

```
template<template<class> class T, class U>
struct eval<T<U> > { /*...*/ };

template<template<class,class> class T, class U, class V>
struct eval<T<U, V> > { /*...*/ };
```

There are several such specializations, up to some predetermined limit. The limit is arbitrary, and templates with higher arity than this limit cannot be evaluated by the MPL lambda facility.

3 The Problem: Matching Template Template Parameters

We would like to remove the code repetition and lift the arbitrary hard-coded limit. Unfortunately, the obvious implementation based on variadic templates does not work as one might expect:

```
template<template<typename...> class T, typename... U>
struct eval<T<U...> > { /*...*/ };
```

The normal interpretation of variadic templates is that a template type parameter pack "typename... Args" acts like a set of templates type parameters typename Args1, typename Args2, ..., typename Args$N$ for the appropriate value N. Under this interpretation, the partial specialization above should match, e.g., std::pair<int, float>.

However, this is not the way variadic templates are specified. In the partial specialization, the template template parameter T can only match a class template with the same "type and form" ([temp.arg]p1), which means that it can only match a template whose template prameter list consists of a single template type parameter pack. Thus, std::tuple<int, float> would match this partial specialization, but std::pair<int, float> will not.

Due to this strict rule that the template argument lists of template arguments must exactly match the template argument lists of their corresponding template template parameter, there is no way to use variadic templates to eliminate the code repetition currently needed to handle all forms of eval. Worse, since the addition of variadic templates means that there are more combinations of "type and form" of template arguments that need to be matched, e.g., one would have to add specializations such as:

```
template<template<typename...> class T, typename... U>
struct eval<T<U...> > { /*...*/ };

template<template<class, typename...> class T, class U, typename... V>
struct eval<T<U, V...> > { /*...*/ };

/* and so on ...*/
```

This problem has so far been discovered in Boost.MPL and Boost.Proto, although it will affect any library that is based on pulling apart arbitrary types into templates and their template arguments.

4 Proposed Solution

The behavior of variadic templates with respect to template template parameters limits the expressiveness of variadic templates and can be surprising to users. We propose to loosen the matching rule for variadic template template parameters to the "intuitive" formulation described above. Given a template template parameter P whose template parameter list terminates with a template parameter pack, P can match any template argument A with a template parameter list that is identical to P's template parameter list up to the template parameter pack, followed by template parameters that match the template parameter pack in type and kind (but do not necessarily have to be template parameter packs themselves). Thus, given our eval partial specialization

```
template<template<typename...> class T, typename... U>
struct eval<T<U...> > { /*...*/ };
```

we say that the template template parameter T can match any of the following templates:

```
template<typename...> class A;
template<typename T> class B;
template<typename T1, typename T2> class C;
template<typename T1, typename T2, typename...> class D;
```

With this (relatively minor) change to the formulation of variadic templates, we were able to express the relevant parts of MPL and Boost.Proto using variadic templates, without the need for code repetition. We have implemented this change within the GNU C++ compiler (which required less than a day of effort) and verified that (1) it addresses the stated problems, and (2) it did not break any existing uses of variadic templates within the GNU C++ Standard Library implementations of Library Technical Report 1. Since variadic templates is a new feature in C++0x (and no released compiled supports them), we don't expect that changing these semantics will have any effect on users.

5 Proposed Wording

This proposed wording introduces the change described in the previous section, along with a few minor tweaks that are necessary for this change to be useful in real libraries.

5.1 Templates [temp]

5.1.1 Template arguments [temp.arg]

8 A *template-argument* followed by an ellipsis is a pack expansion (14.5.3). ~~A *template-argument* pack expansion shall not occur in a *simple-template-id* whose *template-name* refers to a class template unless the *template-parameter-list* of that class template declares a template parameter pack.~~

5.1.2 Template template arguments [temp.arg.template]

3 A *template-argument* matches a template *template-parameter* (call it P) when each of the template parameters in the *template-parameter-list* of the *template-argument*'s corresponding class template or template alias (call it A) matches the corresponding template parameter in the *template-parameter-list* of P. When P's *template-parameter-list* contains a template parameter pack ([temp.variadic]), the template parameter pack will match zero or more template parameters or template parameter packs in the *template-parameter-list* of A with the same type and form as the template parameter pack in P (ignoring whether those template parameters are template parameter packs)[*Example*:

```
template<typename T> struct eval;

template<template<typename, typename...> class TT, typename T1, typename... Rest>
struct eval<TT<T1, Rest...> > { };

template<typename T1> struct A;
template<typename T1, typename T2> struct B;
template<int N> struct C;
template<typename T1, int N> struct D;
template<typename T1, typename T2, int N = 17> struct E;

eval<A<int>> eA; // okay: matches partial specialization of eval
eval<B<int, float>> eB; // okay: matches partial specialiation of eval
eval<C<17>> eC; // error: C does not match TT in partial specialization
eval<D<int, 17>> eD; // error: D does not match TT in partial specialization
eval<E<int, float>> eE: // error: E does not match TT in partial specialization
```

– end example]

5.1.3 Template argument deduction [temp.deduct]

2 When an explicit template argument list is specified, the template arguments must be compatible with the template parameter list and must result in a valid function type as described below; otherwise type deduction fails. Specifically, the following steps are performed when evaluating an explicitly specified template argument list with respect to a given function template:

- The specified template arguments must match the template parameters in kind (i.e., type, non-type, template). There must not be more arguments than there are parameters unless at least one parameter is a template parameter pack, and there must not be fewer arguments than there are parameters. Otherwise, type deduction fails.

- All references in the function type of the function template to the corresponding template parameters are replaced by the specified template argument values. If a substitution in a template parameter or in the function type of the function template results in an invalid type, type deduction fails. [Note: The equivalent substitution in exception specifications is done only when the function is instantiated, at which point a program is ill-formed if the substitution results in an invalid type.] Type deduction may fail for the following reasons:

  – Attempting to instantiate a pack expansion containing multiple parameter packs of differing length.

References

[1] D. Gregor, J. Järvi, J. Maurer, and J. Merrill. Proposed wording for variadic templates (revision 2). Technical Report N2242=07-010, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2007.

[2] D. Gregor, J. Järvi, and G. Powell. Variadic templates (revision 3). Number N2080=06-0150 in ANSI/ISO C++ Standard Committee Pre-Portland mailing, October 2006.

[3] A. Gurtovoy. The Boost MPL library. `http://www.boost.org/libs/mpl/doc/index.html`, July 2002.