Doc No: SC22/WG21/N1406
J16/02-0064

Date: October 21, 2002

Project: JTC1.22.32

Reply to: Herb Sutter
Microsoft Corp.
1 Microsoft Way
Redmond WA USA 98052
Fax: +1-928-438-4456
Email: hsutter@gotw.ca

Proposed Addition to C++: Typedef Templates

1. The Problem, and Current Workarounds

We would like to allow the programmer to create a synonym for a template where some, but not all, actual template arguments are fixed.

The problem is important because such a facility would make it possible to create more easily usable template libraries. For example, consider a template like this:

```
template<typename T1,
         typename T2,
         typename T3 = int
         typename T4 = string>
class C { /*...*/ };
```

Today, default template arguments already enable programmers to use the template more naturally (and less tediously) as just:

```
C<bool, short> x; // synonym for C<bool, short, int, string>
```

Alternatively, we can also use `typedefs` to create a synonym for another type, including a synonym for a template specialization with all actual template arguments specified:

```
typedef C<bool, short, long, wstring> Phil;
Phil p;
```

It is not, however, possible in general to use default arguments or typedefs to create a more usable name for a template where some, but not all, actual template arguments are fixed. The ability to create a synonym which specifies only some template arguments while allowing others to still vary would be useful and help to create more naturally usable names in libraries.

In existing practice, including in the standard library, type names nested inside helper class templates are used to work around this problem in many cases. The following is one example of this usual workaround; the main drawback is the need to write `::Type` when using the typedef'd name.

```
template< typename T >
struct SharedPtr
{
  typedef Loki::SmartPtr
    <
      T,                 // note, T still varies
      RefCounted,        // but everything else is fixed
      NoChecking,
      false,
      PointsToOneObject,
      SingleThreaded,
      SimplePointer<T>  // note, T can be used as here
    >
    Type;
};

SharedPtr<int>::Type p; // sample usage, "::Type" is ugly
```

What we'd really like to be able to do is simply this:

```
template< typename T >
typedef Loki::SmartPtr
  <
    T,                   // note, T still varies
    RefCounted,          // but everything else is fixed
    NoChecking,
    false,
    PointsToOneObject,
    SingleThreaded,
    SimplePointer<T>    // note, T can be used as here
  >
  SharedPtr;

SharedPtr<int> p;       // sample usage, "::Type" is ugly
```

For another example, the standard library's `rebind` helpers fall into this category:

```
template<typename T> class allocator { //...
  template<typename U>
  struct rebind { typedef allocator<U> other; };
};

allocator<T>::rebind<U>::other x; // sample usage
```

What we'd really like to be able to do is simply this:

```
template<typename T> class allocator { //...
  template<typename U>
  typedef allocator<U> rebind;
};
```

```
allocator<T>::rebind<U> x;        // sample usage
```

In fact, the standard itself says: “The template class member rebind […] is effectively a template typedef: if the name Allocator is bound to SomeAllocator<T>, then Allocator::rebind<U>::other is the same type as SomeAllocator<U>.” [emphasis mine]

The workaround is ugly, and it would be good to replace it with first-class language support that offers users a natural C++ template syntax.

This proposal fits into the categories of:

- remove embarrassments (the absence of typedef templates is a known weakness in the language)
- make the language easier to learn (eliminating the needless “`::Type`” wart makes the syntax more uniform for users)
- improve support for library building
- improve support for generic programming

2. The Proposal

In brief, this proposal is to allow typedef templates having exactly the same semantics as the workaround. That is, the following general typedef template:

```
template< /* stuff */ > typedef /* something */ X;

// usage: X< /* whatever */ >
```

is exactly equivalent to the following workaround, except that users need not write `::Type` when using the typedef’d name:

```
template< /* stuff */ >
class X {
public:
  typedef /* something */ Type;
};

// usage: X< /* whatever */ >::Type
```

As with the workaround, redeclarations of typedef templates should obey the ODR.

2.1 Basic Cases

A typedef introduces a synonym, rather than a complete new type. Similarly, a typedef template introduces a parameterized synonym, not a complete new type. One purpose for allowing templatization of a typedef is to introduce a simplified synonym for an existing template where some but not all template arguments are fixed. For example:

```
template<typename A, typename B> class X { /* ... */ };

template<typename T> typedef X<T,int> Xi;
```

```
Xi<double> Ddi;    // equivalent to X<double,int>
```

A typedef template can be modeled like a partial specialization, with the definition being the primary class template. The syntax naturally follows the existing syntax for function and class templates:

```
void f( int ); // function
template<typename T> void f( T );
               // function template, usage f<int>

class X { };   // class
template<typename T> class X { };
               // class template, usage X<int>

typedef map<string, Employee> EmployeeRegistry; // typedef
template<typename T> typedef map<string, T> Registry;
               // typedef template, usage Registry<Employee>
```

It uses the same rules as function and class templates for dependent names (including the use of typename within the typedef template for dependent type names), non-type parameters, and template template parameters.

Here’s an example that comes up in many class templates, particularly in policy-based design (heavily used in Loki) where there are many template parameters and we currently can’t express a typedef name that fixes some but not all of the types. In this example, I cite Loki’s SmartPtr, which is very flexible because it allows customization via several policy template parameters. Unfortunately, having so many template parameters also makes it harder to use. There are several common uses of Loki’s SmartPtr with particular template parameters fixed that it would be useful to be able to invoke more simply via a synonym. For example:

```
template< typename T >
typedef Loki::SmartPtr
    <
      T,                  // note, T still varies
      RefCounted,         // but everything else is fixed
      NoChecking,
      false,
      PointsToOneObject,
      SingleThreaded,
      SimplePointer<T>    // note, T can be used as here
    >
    SharedPtr;

template< typename T >
typedef Loki::SmartPtr
    <
      T,
      RefCounted,
      NoChecking,
      false,
      PointsToArray,
      SingleThreaded,
      SimplePointer<T>
```

```
        >
        SharedArray;

    template< typename T >
    typedef Loki::SmartPtr
        <
          T,
          NonCopyable,
          NoChecking,
          false,
          PointsToOneObject,
          SingleThreaded,
          SimplePointer<T>
        >
        ScopedPtr;

    template< typename T >
    typedef Loki::SmartPtr
        <
          T,
          NonCopyable,
          NoChecking,
          false,
          PointsToArray,
          SingleThreaded,
          SimplePointer<T>
        >
        ScopedArray;
```

2.2 The Main Choice: Specialization vs. Everything Else

After discussion on the reflectors and in the Evolution WG, it turns out that we have to choose between two mutually exclusive models:

1. *A typedef template is not itself an alias; only the (possibly-specialized) instantiations of the typedef template are aliases.* This choice allows us to have specialization of typedef templates.
2. *A typedef template is itself an alias; it cannot be specialized.* This choice would allow:
   - deduction on typedef template function parameters (see 2.4)
   - a declaration expressed using typedef templates be the same as the declaration without typedef templates (see 2.5)
   - typedef templates to match template template parameters (see 2.6)

   Note that none of these three items is possible using the current workaround, either.

This paper proposes Option 1, thus favoring specialization at the expense of matching, deduction, and same-declarations, which are not possible using the current workaround either.

2.3 Specialization

Consider the following typedef template:

```
template<typename A, typename B> class X { /* ... */ };

template<typename T> typedef X<T,int> Xi;
```

To specialize the typedef template, use the same syntax as when specializing class and function templates:

```
// specialization for string
template<> typedef UnrelatedType Xi<string>;

...

Xi<double> Ddi;   // uses base template

Xi<string> Ssi;   // uses specialization
```

To partially specialize the typedef template, use the same syntax as when partially specializing class and function templates — the only trick is to remember where the template argument list goes, namely right after the name being specialized. For class templates, the standard says: "For partial specializations, the template argument list is explicitly written immediately following the class template name." So, for partial specializations of typedef templates, the template argument list is explicitly written immediately following the typedef template name:

```
// partial specialization for pointers
template<typename T> typedef AnotherUnrelatedType<T> Xi<T*>;

...

Xi<double> Ddi;   // uses base template

Xi<int*> Ipi;     // uses partial specialization
```

Here are additional motivating cases for allowing specialization, provided by Peter Dimov:

```
template<int> typedef int int_exact;
template<> typedef char int_exact<8>;
template<> typedef short int_exact<16>;
// ...

template<class T> typedef T remove_const;
template<class T> typedef T remove_const<T const>;
```

2.4 Different Declarations

Because typedef templates are semantically a direct equivalent for the workaround, they are not deducible when used as parameter types in function declarations.

```
template<typename T>
typedef T MyT;

template<typename T>
void f( MyT<T> );
void g() {
  MyT<int> val = 42;
  f( val );        // 1
  f( 42 );         // 2
}
```

In an alternative proposal (one that did not allow specialization), `f()` would would be deducible in both lines 1 and 2 above.

2.5 Different Declarations

A declaration having a parameter whose type is expressed in terms of a typedef template is not identical to the same declaration with the parameter expressed in terms of the type for which the typedef template is a synonym.

Some feel it would be desirable to have such declarations be identical, however. Although this paper does not propose that, a motivating example follows, provided by Peter Dimov:

```
template<class T, class P> class smart_ptr;
template<class T> typedef smart_ptr<T, SharedPolicy> shared_ptr;

template<class T> void f(smart_ptr<T, SharedPolicy>);
template<class T> void f(shared_ptr<T>);
```

In this proposal, the last two lines declare different templates.

2.6 Non-Matching of Template Template Parameters

To support allowing typedef templates to match template template parameters while still allowing typedef template specialization, we would have to add rules outlined in Santa Cruz by John Spicer that add complexity and would still yield surprising results. The discussion in Santa Cruz is that the effort is probably not worth it if this would become the only exception to the "syntactic synonym for the workaround" formulation of typedef templates, which is the formulation proposed in this paper.

Some feel it would be desirable to be able to match template template parameters, however. Although this paper does not propose that, a motivating example follows:

```
template<template<class> class X> class Y {};

template<class, class> class Z {};

template<class T> typedef Z<T, T> A;

Y<A> a; // uses adapted Z
```

3. Interactions and Implementability

3.1 Interactions

The proposed feature is intended to be a natural application of existing template syntax to the existing `typedef` keyword. Interactions with the rest of the language are limited because typedef templates do not create a new type or extend the type system in any way; they only create synonyms for other types. Further, the proposed feature is an exact equivalent of an existing workaround having well-understood semantics.

This is not a one-off or special-purpose feature. Consider the example in §2.1, and the ease of use of letting the programmer write:

```
SharedPtr<int> p;

SharedArray<int> a;
```

instead of:

```
SharedPtr<int>::Type p;

SharedArray<int>::Type a;
```

or, worse still:

```
SmartPtr<int, RefCounted, NoChecking, false,
         PointsToOneObject, SingleThreaded,
         SimplePointer<int> > p;

SmartPtr<int, RefCounted, NoChecking, false,
         PointsToArray, SingleThreaded,
         SimplePointer<int> > a;
```

The naturalness and ease of use of the first case is possible only with typedef templates, and will make advanced C++ libraries more accessible to programmers.

3.2 Implementability

A sample implementation that allows the basic usages, but not specialization, was created as an unshipped extension within the Microsoft compiler with little difficulty. The work to add specialization and deduction is not expected to be difficult, but of course that won't be known for sure until it's done.