Doc No: SC22/WG21/N1449
J16/03-0032

Date: April 07, 2003

Project: JTC1.22.32

Reply to: Gabriel Dos Reis
INRIA Sophia Antipolis
06902 Sophia-Antipolis France
Fax: 334 92 38 79 78
Email: gdr@acm.org

Mat Marcus
Adobe Systems, Inc.
801 North 34th Street
Seattle, WA 98103 USA
Fax: 206.675.7825
Email: mmarcus@emarcus.org

Proposal to add template aliases to C++

1. The Problem

Today, many libraries are implemented with very general class or function templates. This trend offers users a high degree of control and flexibility. But these benefits are not without cost. For example, a user might need to supply multiple template parameters, but sometimes a simpler template is desired (say one requiring only a single template parameter). A common idiom is to supply another template containing a nested typedef that simulates a so-called 'typedef template' or 'meta-function'. For more details see N1406. The reader is encouraged to have a copy in hand while reading this proposal for a general explanation of the issues. In this paper we will focus on describing an aliasing mechanism that allows the two semantics mentioned in N1406 to coexist instead being regarded as mutually exclusive. First let's consider a toy example:

```
template <typename T>
class MyAlloc {/*…*/};

template <typename T, class A>
class MyVector {/*…*/};

template <typename T>

struct Vec {
   typedef MyVector<T, MyAlloc<T> > type;
};

Vec<int>::type p; // sample usage
```

The fundamental problem with this idiom, and the main motivating fact for this proposal, is that the idiom causes the template parameters to appear in non-deducible context. That is, it will not be possible to call the function `foo` below without explicitly specifying template arguments.

```
template <typename T> void foo (Vec<T>::type&);
```

Also, the syntax is somewhat ugly. We would rather avoid the nested ::type call. We'd prefer something like the following:

```
template <typename T>
using Vec =  MyVector<T, MyAlloc<T> >; //defined in section 2 below

Vec<int> p;       // sample usage
```

Note that we specifically avoid the term "typedef template" and introduce the new syntax involving the pair "using" and "=" to help avoid confusion: we are not *defining* any types here, we are introducing a synonym (i.e. alias) for an abstraction of a *type-id* (i.e. type expression) involving template parameters. If the template parameters are used in deducible contexts in the type expression then whenever the template alias is used to form a *template-id,* the values of the corresponding template parameters can be deduced – more on this will follow. In any case, it is now possible to write generic functions which operate on `Vec<T>` in deducible context, and the syntax is improved as well. For example we could rewrite `foo` as:

```
template <typename T> void foo (Vec<T>&);
```

We underscore here that one of the primary reasons for proposing template aliases was so that argument deduction and the call to `foo(p)` will succeed.

Which of the categories that we're interested in addressing does this fit into?

improve support for library building -- Yes

improve support for generic programming -- Yes

make C++ easier to teach and learn -- Yes

remove embarrassments -- Yes

2. The Solution

This proposal introduces the ability to declare a template name that acts as an alias. Consider:

```
template <typename T>
using Vec =  MyVector<T, MyAlloc<T> >;
```

The above can be read as: declare `Vec` as a "template alias" (synonym) for the "source template" `MyVector<T,MyAlloc<T> >`. That is, `Vec` does not introduce any `typedefs`. Rather it creates what can be considered a mapping from template parameters to type expressions, possibly involving the template parameters. If the source template parameters could be used in deducible context in a corresponding function template signature then `Vec` can be used in its place. In this case the corresponding function would look like this:

```
template <class T>
void corresponding(MyVector<T, MyAlloc<T> >);
```

A good mental model is: `T` will be deducible in the context of `foo` exactly when `T` is deducible in the context of the `corresponding` function.

2.1 Specialization

It has been claimed that it is not possible to enjoy the benefits of deducibility and specializability. It is true that to maintain deducibility we are disallowing specialization of the template alias. But a key

point is that specialization is still possible as a by-product of existing language construct (e.g. traits) to redirect the template alias to specific specializations as illustrated below. Say for example that we wish to specialize `Vec` for pointer types. Rather than trying to specialize the template alias to `Vec<T*>` we specialize the source template to `MyVector<T*, A>`:

```
template <class T, class A>
class MyVector<T*, A> {…};
```

Then `Vec` will pick up this specialized behavior from `MyVector`, and deducibility is retained. Some cases are more difficult to tackle. Consider the `int_exact` example, expressed in the "typedef template" language of N1406:

```
template<int> typedef int int_exact;     // Not from this proposal…
template<>    typedef char int_exact<8>;
template<>    typedef short int_exact<16>;
// …
```

With a little more work this can be expressed in the language of template aliases. First we must create a "traits" source template that allows us to capture the `int_exact` specializations in a template, following the idiomatic style presented at the beginning of this paper.

```
template<int> struct int_exact_traits    { typedef int type; };
template< >   struct int_exact_traits<8> { typedef char type; };
template< >   struct int_exact_traits<16>{ typedef short type; };
// …
```

Then we can define a template alias for the `int_exact_traits` class:

```
template <int N>
using int_exact = int_exact_traits<N>::type;
```

In this case we retain the exact semantics (and constraints) as in N1406. That is,

```
template <int N>
void foo(int_exact<N>&);

int_exact<8> i8;

foo(i8); // deduction fails, the above foo is not callable
```

One can sum up the specialization issues as follows. Specializations of the template (alias) name can be achieved through the use of traits that specialize the "initializing" type expressions of the template alias . That is, if the source template parameter set can be deduced in the corresponding function template context then so can, then those template parameters can be deduced for function calls involving the template alias. Finally, it is possible to trade away deducibility when needed, as in the `int_exact`/traits example. This requires more effort using template aliases then it does using the N1406 proposal, but it offers the same power and ease of use for the client. For the above reasons we extend the power of specialization offered by N1406 with the power to use template aliases in deducible context.

Interactions and Implementability

We discussed specialization and deducibility above. Another interesting area of interaction is with namespaces and ADL. Imagine that we declare the `Vec` template alias inside namespace `N`, while let us

assume that `MyVector` and `MyAllocator` are defined in namespace `Detail`. One might expect that `Vec's` presence in `N` would cause ADL to look in namespace `N` when looking up a function, say, `f(Vec<int>&)`. But `Vec` is meant to be just an alias, so initially we assume that the presence of the `Vec<int>` argument only contributes namespace `detail` to the lookup namespace set for `f`.

Another point worth noting is that template aliases do not introduce any new types or templates. In the example above, `Vec<T>` is simply another name for `MyVector<T, MyAlloc<T> >`. As a consequence

```
template <class T> void foo(Vec<T>&);
```

is a redeclaration of

```
template <class T> void foo(MyVector<T, MyAlloc<T> >&);
```

This issue is also discussed in N1406 section 2.5.

Note: it has not escaped our attention that a nullary template alias is essentially equivalent to a traditional typedef, but we have found no compelling reason to include a discussion about it (nor is there an obvious workable syntax).

4. Acknowledgements

Thanks to Dave Abrahams for contributions towards how specialization works in the context of a template alias.

5. References

N1406: Proposed addition to C++: typedef templates, Herb Sutter

6. Afterword

The original draft of this proposal was well received by the Evolution Working Group at the Oxford meeting. After the initial presentation of this paper we worked on carrying these ideas further. First we considered extending the proposal to include the notion of function template aliases:

```
template <class T>
using F = f<T, MyAlloc<T> >(int, char); // does this require a signature?
```

We also briefly considered non-template aliasing:

```
using F = f(int);
using Cos = cos; // whole overload set?
```

Two straw polls were taken regarding syntax. A strong majority voted to avoid the typedef template syntax, in favor of the "=" syntax. A second vote indicated strong preference for the "using" keyword as opposed to a word like "alias" or the absence of any keyword as in the draft version of this proposal. The motivation for using any keyword at all stemmed partly from the desire to use a syntax that might be compatible with the non-template aliasing direction briefly outlined above.

The core ideas of template aliasing originate from message `c++std-ext-5658` which generated a fruitful discussion and suggestions like David Vandevoorde's "template alias" name and syntax `template<typename T> Vec = vector<T, MyAlloc<T> >;`