Doc. no. WG21/N1489=J16/03-0072
Date: 04 July 2003
Reply-To: Bjarne Stroustrup
Texas A&M University
College Station
Texas 77843-3112
Email: bs@cs.tamu.edu
and
Gabriel Dos Reis
INRIA Sophia Antipolis, BP 79
2004 route des Lucioles
06902 Sophia-Antipolis — France
Email: gdr@acm.org

Templates aliases for C++

This note is an expansion on the proposal [2] to add template aliases to C++. It further discusses the fundamental issue, the solution suggested in [2] and several possible directions of generalizations to non-template and non-type settings. We encourage the reader to have copies of the papers [2, 3] in hand while reading this note. This does not constitute a final wording for inclusion in the working paper; rather it is a collection of ideas as a basis for further discussions.

1 Parameterized type aliases

C and C++ provide `typedef` as a mechanism for introducing a new name for an existing type. With increasing use of parameterized types in C++ libraries and programs, the need for a notion of parameterized type aliases has been pressing. The famous standard allocator rebind hack is a well-known simulation of parameterized type aliases and work-around for lack of language feature. The notion of *template alias* [2] aims at providing support for declaring families of aliases. This new feature gives opportunities for building advanced libraries as those described by Walter Brown [1], otherwise hard or impossible to achieve within current standard C++ semantics.

With current C++, parameterized aliases for type expressions arew simulated along the lines of the following example:

--- Example #1 ---

```
// My own storage allocator
template<class T>
  struct MyAllocator {
```

```
    // ...
  };

// An intermediate class as a means to introduce
// an alias for std::vector<T, MyAllocator<T> >
template<class T>
  struct Vec {
    typedef std::vector<T, MyAllocator<T> > type;
  };

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

While in simple usages like above, the idiom works pretty well, it quickly breaks down when used in conjunction with parameterized functions. Consider:

--- Example # 2 ---

```
template<class T>
  void process(typename Vec<T>::type& v)
  { /* ... */ }

int main()
{
   Vec<int>::type v(275);
   process(v);           // ERROR: cannot call ‘‘process’’
   // ...
}
```

The breakage comes from the inability to deduce the template argument list for the function template `process`; see [2, 3] for further details.

Question: Would the problem go away if one could make `T` deducible in the context `Vec<T>::type`? No, that would solve only part of the problem. Examples, such as that standard allocator "rebind hack" would not be dealt with by such deduction. Thus, a general aliasing mechanism is needed.

1.1 Parameterized alias: semantics

As a matter of semantics improvement and syntax simplification, it is suggested in the paper [2] to adopt the notation

```
template<class T>
  using Vec = std::vector<T, MyAllocator<T> >;
```

with the following meaning:

- it declares `Vec` as the name of a template of one argument;

- given a template-argument `A`, the template-id `Vec<A>` is syntactically equivalent to `std::vector<A, MyAllocator<A> >`.

With those semantics, the core ideas behind examples #1 and #2 may be rephrased as:

```
template<class T>
  using Vec = std::vector<T, MyAlocator<T> >;

template<class T>
  void process(Vec<T>& v)
  { /* ... */ }

int main()
{
   Vec<int> v(297);
   process(v);      // OK: calls ‘‘process<int>’’
   // ...
}
```

Now, the call to the function `process` is satisfactorily resolved because its declaration is equivalent to

```
template<class T>
  void process(std::vector<T, MyAllocator<T> >&);
```

which has a form suitable for successful template-argument deduction. The parameterized type alias feature may be summarized as a mechanism to declare a template-name, which when presented with a template argument list produces an alias for the type expression obtained after substitution of the template arguments into the “initializer”. This name aliasing feature may be generalized in various ways as discussed in §2.3.

1.2 Parameterized alias: syntax

The semantics of the proposed extension seem to cover the fundamental issues as identified in the papers [1, 2, 3]. However, the syntax may look a bit contorted. The declaration

```
template<class T>
  using  Vec =  std::vector<T, MyAllocator<T> >;
```

roughly speaking, contains five (5) parts:

1. the template parameters declaration `template<class T>`,
2. the keyword `using`,
3. the newly-declared template-name `Vec`,

4. the token =,

5. and the type expression `std::vector<T, MyAllocator<T> >`.

As a general principle, an alias is an alternate name for an entity. So the first, third, and fifth parts are necessary: they state that we are introducing a template with a set of parameters, introducing a new name, and stating the entity being aliased, respectively. The second (`using`) and the fourth (=) parts are syntactic sugar. An alternative choice for a new keyword would be `let` — used in BCPL and many functional languages for variable declaration — or `alias`. However, those are short, probably common identifiers, with a high potential for breaking existing codes.

On the other hand the sentence

```
template<class T>
  using Vec = std::vector<T, MyAllocator<T> >;
```

can be read/interpreted as: *from now on, I'll be using* `Vec<T>` *as a synonym for* `std::vector<T, MyAllocator<T> >`. With that reading, the new syntax for aliasing seems reasonably logical. Therefore, we propose the following formal syntax for the declaration of a parameterized type alias:

`template <` *template-parameter-list* `>` *alias-declaration*

alias-declaration:
`using` *identifier* `=` *type-id*

A key question is: ***does it fit with/generalize existing name aliasing mechanism?*** That topic will be discussed in §2.

1.3 Point of declaration

Since an alias-declaration is a declaration , it is important to know the point of declaration of the alias-name it declares. For reasons discussed below we propose that the point of declaration of a name declared by an alias-declaration be immediately after its *type-id* "initializer". For example, in

```
template<class T>
  using Vec = std::vector<T, MyAllocator<T> >;
```

the point of declaration of `Vec` is just before the semicolon terminating the declaration. In particular, the following is ill-formed

```
template<class T>
  using List = std::pair<T, List<T>*>;
```

The reason is that `std::pair<T, List<T>*>` is not an existing type since there is no declaration for `List`.

The core of the rule determining the point of declaration for names introduced by an alias-declaration is not an innovation. It is similar to existing rule that governs the point of declaration of an enumerator. Indeed, current Standard text says (3.3.1/3)

> The point of declaration for an enumerator is immediately after its *enumerator-definition*

That rule was intended to be adapted for namespace-aliases (quote William Miller); at least current Standard makes

```
namespace N = N;
```

ill-formed because there is no existing namespace that the name `N` (on the right hand side) resolves to.

1.4 Specializations

Ability to specialize template aliases is a topic that has been discussed a lot in most papers about "template typedef" or "template aliases". There are evidences that it is a useful feature that we should provide along with template aliases. Herb Sutter [3] gave what we consider compelling examples, among which the following: Naming a sized-integer type, the size of which is expressed as a template argument (obviously useful in generic programming context). Borrowing syntax from that paper, we would have:

```
template<int> typedef int int_exact;     // default alias
template<>    typedef char int_exact<8>;
// ...
```

As pointed out in the paper [2], the above constructs can be expressed with the template alias mechanism discussed in §1.1 and §1.2 with the aid of an auxiliary traits class:

```
template<int>
  struct int_exact_traits {
     typedef int type;
  };
// create specializations
template<>
  struct int_exact_traits<8> {
     typedef char type;
  };
// and so on...

// create the alias
template<int N>
  using int_exact = typename int_exact_traits<N>::type;
```

A fundamental concern about this approach is: ***Is the intermediate traits class necessary?*** We believe "yes" because an alias is an alternate name for an entity and the auxiliary class `int_exact_traits` is just designating the entity we want to create an alias for. Providing a shorthand that does not introduce a second name would leave a single name denoting both an alias and the entities it is an alias for. That has a high potential for creating confusion.

Therefore, we propose that a template-name introduced by an alias-declaration should not be specialized. If an alias could be specialized, it would be difficult — potentially impossible — to resolve combinations of specializations of the template itself (which of course must be allowed) and specializations of the alias.

2 General alias declaration

In this section, we would like to explore the extent to which the newly proposed name aliasing scheme fits and generalizes the existing aliasing mechanisms, such as typedef and namespace alias. Firstly, we will compare the parameterized type alias declaration as suggested in §1.2 with existing existing template declaration.

Let us recall that the grammar of a template declaration is

template-declaration:
`export`$_{opt}$ `template <` *template-parameter-list* `>` *declaration*

It is apparant that if we slightly extend the standard *block-declaration* production rule with an *alias-declaration* rule as follows

block-declaration:
simple-declaration
asm-definition
namespace-alias-definition
using-declaration
using-directive
alias-declaration

alias-declaration:
`using` *identifier* `=` *type-id*

then the parameterized type alias syntax neatly fits the general syntax. It thus remains to elaborate on the new rule.

An alias-declaration is a declaration, and not a definition. An alias-declaration introduces a name into a declarative region as an alias for the type designated by the right-hand-side of the declaration. *The core of this*

proposal concerns itself with type name aliases, but the notation can obviously be generalized to provide alternate spellings of namespace-aliasing or naming set of overloaded functions (see §2.3 for further discussion). It may be noted that the grammar production *alias-declaration* is acceptable anywhere a `typedef` declaration or a *namespace-alias-definition* is acceptable.

2.1 Alternate spelling for `typedef`

A `typedef` declaration can be viewed as a special case of a non-template alias-declaration. For example, the declaration

```
typedef double (*analysis_fp)(const vector<Student_info>&);
```

could be rewritten in the new alias-declaration syntax as

```
using analysis_fp = double (*)(const vector<Student_info>&);
```

The new syntax can be considered an improvement over the existing non-linear syntax, in that it visually separates the declared name from the spelling of the type an alias is being declared for.

Because a non-template alias-declaration that aliases a *type-id* is semantically equivalent to a `typedef` declaration, we suggest that all rules applicable to *typedef-name*s be applicable to *alias-name*s when the alias denotes a type.

Since allowing this new syntax does not introduce a new possibilities for aliasing it cannot lead to problems with naming that programmers don't currently face. We propose to allow the new syntax for generality.

2.2 Alternate spelling for namespace-aliasing

The alias-declaration syntax may also be generalized to cover the namespace alias definition syntax:

```
using WCL = WorldCompanyLibrary::ThirdRevision;
```

would be another of way saying

```
namespace WCL = WorldCompanyLibrary::ThirdRevision;
```

The grammar rule for *alias-declaration* would just need to be extended as indicated below

alias-declaration:
`using` *identifier* `=` *type-id*
`using` *identifier* `=` *qualified-namespace-specifier*

The issue of whether to allow extending a namespace through its alias or not should be considered seperately from this proposal; however, it worths pointing out because the alias is syntactically equivalent to the aliased namespace, the general new name aliasing syntax nicely fits that eventual language extension.

The alias-declaration syntax appears to be more general and more uniform. It also contains the traditional using-declaration as a special case.

2.3 Interaction with using-declaration

It is possible to generalize the notion of alias beyond types and namespaces to functions, variables, etc. We do not see sufficient benefits from doing this and can imagine serious overuse leading to confusion about which functions and variables are used. Consequently, we do not propose the generalizations mentioned in this section. Furthermore, we do not plan to work further on these generalizations unless someone comes up with examples that indicate significant usefulness.

In current C++, a *using-declaration* brings into scope a name declared elsewhere (in a any named scope). Within the framework of the proposed name aliasing syntax, the construct

```
using std::ostream;
```

would just be a short-hand for

```
using ostream = std::ostream;
```

where the new-to-be introduced identifier `ostream` is the same as the unqualified part of the name (`std::ostream`) we're creating an alias for. In that respect, the alias-declaration generalizes the existing using-declaration.

Allowing just this would be non-problematic and provide no new functionality. We must consider the case where we introduce a new name for an entity. For example:

```
using mystream = std::ostream;
```

This extends the alias-declaration scheme to non-type and non-namespace names. For example:

```
int i;
using r = i;
```

This would provide a way to name sets of overloaded functions.

```
#include <cmath>
using Cos = std::cos; // whole overload set
using C = std::cos(double); // select double std::cos(double);
```

In the last alias-declaration, the specification of the parameter list `(double)` unambiguously designates which function we are considering in the overload set denoted by `std::cos`. Therefore, there is no need for return type specification. This just highlights the fact that an alias-declaration does not introduce a name as an alias for a declaration; rather it introduces a name as a alternate spelling for an entity, *i.e.* it aliases another name. Similarly, in

```
template<class T>
  using F = f<T, MyAllocator<T> >(int, char);
```

the explicit template-argument list `<T, MyAllocator<T> >` helps to narrow down the overload set designated by `f`, and the parameter list `(int, char)` then designates the desired function.

References

[1] Walter E. Brown, *A Case for Template Aliasing*, document no. WG21/N1451=J16/03-0034.

[2] Gabriel Dos Reis and Mat Marcus, *Proposal to add template aliases to C++*, document no. WG21/N1449=J16/03-0032.

[3] Herb Sutter, *Typedef templates*, document no. WG21/1406.