Template deduction for nested classes | P0293R0

S. Davis Herring*

October 14, 2016

Audience: EWG

1 Abstract

Because type names nested inside a dependent type may be *typedef-names*, template deduction is not attempted on them. However, nested classes in a class template can be referenced only with names of that form and thus may never be deduced. This restriction is severe in contexts like partial specialization and conversion function templates where template arguments must be deduced. This paper proposes to shrink this non-deduced context to allow nested classes to be deduced, but to retain the part necessary to avoid ill-posed deduction problems. Code like the following is then possible:

```
template<class T> struct C {
  struct iterator { /* ... */ };
  iterator begin();
  // ...
};
template<class T> struct Q {
  using type=int;
  type count();
  // ...
};

template<class T> void fC(typename C<T>::iterator);
template<class T> void fQ1(typename Q<T>::type);
template<class T> void fQ2(typename Q<T>::type,T);

void f() {
  C<int> c;
  Q<int> q;
  fC(c.begin());            // invalid in C++14; calls fC<int> here
  Q<int>::type i=q.count(); // "Q<T>::type", but that's just int!
  fQ1<int>(i);              // no deduction: same meaning as in C++14
  fQ2(i,42);                // deduce from 42: same meaning as in C++14
}
```

2 Problem

It is currently impossible to deduce `T` in the call to `fC` because a *nested-name-specifier* is always a non-deduced context[1]. Obviously the example's intent is trivial to understand; the reason for forbidding deduction there

*herring@lanl.gov

[1] 14.8.2.5 [temp.deduct.type] paragraph 5.1

has to do with *typedef-names* instead, as in `Q`. The mapping from `T` to `Q<T>::type` is non-invertible: even though the argument to `fQ1` is `int`, there is no *unique* `T` to use. In general the inverse mapping is undecidable.

3 Context

The restriction is particularly burdensome for constructor templates and partial specializations, where deduction is required for any use at all. Defining a function template or a partially specialized class template for a set of related nested classes is a natural application, and working around the restriction on doing so is a frequent topic of discussion (see, *e.g.*, `http://stackoverflow.com/questions/12640808/nested-template-and-parameter-deducing`).

There are workarounds available for the case of function templates. The idea from the Barton–Nackman trick can be used to approximate a (non-member) function template:

```
template<class T> struct A {
  struct B {};
  friend void f(B) {}
};
void g() {
  A<int>::B b;
  f(b);         // finds f(A<int>::B) via ADL
}
```

The Curiously Recurring Template Pattern can be used to discover the nested class type (and, with a bit more work, the parameters of the containing instantiation):

```
template<class T> struct C {};
template<class T> struct A {
  struct B : C<B> {
    typedef T outer;
    long x;
  };
};
template<class T> typename T::outer f(C<T> &c) {
  T &t=static_cast<T&>(c);
  return static_cast<typename T::outer>(t.x);
}
int g() {
  A<int>::B b;
  return f(b);    // calls f(C<A<int>::B>&)
}
```

However, neither of these approaches works for partial template specialization, and each requires altering the classes in question (which is of course frequently impractical).

Instead of a *typedef-name* `A<T>::type`, the programmer may and should instead use the real (dependent) type name if one exists. (The substitution cannot be automatic precisely because `T` is unknown.) However, the only form of class name that supports deduction is `X<T,...>` for some template `X` (that is not itself an alias template)[2]; nested classes lack any name of that form! Unfortunately, even when a class template defines a normal nested class, the possibility remains that a specialization of it might introduce an indirect type definition. It therefore may seem that there is no *safe* way to allow deduction in this context.

4 Restricted solution

However, when deduction is performed, an actual type is always available. In considering whether a parameter type like `C<T>::iterator` might match the actual type, it is known whether the actual type is a nested

[2] 14.8.2.5 [temp.deduct.type] paragraph 8

class. If it is not, or if its (unqualified) name is not “iterator”, then deduction is not attempted (and thus does not fail); otherwise, deduction may proceed using the type denoted by the *nested-name-specifier* as the parameter type and the actual type’s containing class as the (new) actual type.

This procedure works even if the member class is an instantiation of a member template: `C<T>::iterator<U>` may be matched by checking that the actual type is an instantiation of a member class template named “iterator”, and if so doing deduction on `C<T>` against the containing class and on `<U>` against the template arguments of the actual type.

Alias templates that expand to the name of a nested class (or instantiation of a member class template) can also be treated; they cannot be specialized, so they may be expanded in the declaration before attempting deduction. Finally, the simpler case of `T::iterator` can be handled trivially.

5 Practicalities

This change may affect existing code by causing additional function template overloads to become applicable or by allowing deduction failures from the additional deduced contexts. An example of the latter, where an additional context provides a result that conflicts with that of a context already supported:

```
template<class T> struct A {struct B {};};
template<> struct A<char> {using B=A<int>::B;};
template<class T> void f(T,typename A<T>::B);
void g() {
  A<int>::B b;
  f('c',b);  // error: T could be char or int
}
```

In C++14, there is no way to deduce `T` as `int`, so the deduction as `char` succeeds. The call also succeeds because, as it happens, the type of `b` is that indicated by the *typedef-name* `A<char>::B`.

To preserve the meaning of as many programs as possible, it is important that deduction not fail entirely in a case beyond even these relaxed rules. Rather, such usage simply remains a non-deduced context. (This failure avoidance is similar to that in 14.8.2.5 [temp.deduct.type] paragraph 5.5.1.) However, newly attempted deductions may nontheless fail according to existing rules, for example because of *e.g.*, 14.8.2.5 [temp.deduct.type] paragraph 17:

```
template<int i> struct A {struct B {};};
template<short s> struct S {};
template<short s> void f(typename A<s>::B,S<s>);
void g() {
  A<1>::B b;
  S<1> s;
  f(b,s);    // error: can't deduce short s=1 from A<int=1>
}
```

In C++14, no attempt is made to deduce `s` from the first argument. (It might be preferable in general to have such cases produce no template argument values rather than cause the entire deduction to fail.)

Finally, referring to a nested class via a member *typedef-name* with the *same* unqualified name can cause a deduction failure:

```
template<class T> struct A {struct B {};};
template<class T> struct Q {using B=typename A<T>::B;};
template<class T> void f(typename Q<T>::B);
void g() {
  A<int>::B b;
  f(b);        // error: no deduction for Q<T>/A<int>
}
```

This failure could be avoided by further complicating the non-deduced context, more or less stating that the context is deduced if and only if deduction succeeds in it. It can be stated much more simply in terms of the idea that deduction should be able to produce no result without failing *per se*.

Because the syntax for a nested class is identical to that for a member *typedef-name*, allowing deduction for one may mislead programmers into thinking it allowed for the other (which remains inadvisable!). Therefore it would be beneficial for implementations to produce warnings when the dependent name in question is a *typedef-name* (or not defined at all). However, since in some cases the template arguments can be supplied explicitly, and forthcoming specializations may introduce nested classes, the program is not ill-formed because of the use of such a dependent name, even when (as for partial specialization) there appears at the point of declaration to be no way for it to be used.

6 Wording

Based on N4604.

To be resolved: 14.8.2.1 paragraph 4.3 may need to be updated to generalize "`P` has the form *simple-template-id*".

Change 14.8.2.5 paragraph 5.1 to read

> The *nested-name-specifier* of a type that was specified using a *qualified-id*, unless the actual type is a nested class (or, if the *unqualified-id* is a *template-id*, an instantiation of a member class template) whose name is the same as that used in the trailing *unqualified-id*.

Change the first and last examples in paragraph 6 to read

> If a type is specified as `AX<T>::BY<T2>`, both `T` and `T2` are non-deduced, unless `A` is a specialization of a member class template named `Y`.
>
> If a type is specified as `void f(typename A<T>::B, A<T>)`, the `T` in `A<T>::B` is non-deduced but the `T` in `A<T>` is deduced even if that in `A<T>::B` is non-deduced.

Add to paragraph 8 the forms

> *`nested-name-specifier`*`::`*`name`*
> *`nested-name-specifier`*`::`*`name`*`<T>`
> *`nested-name-specifier`*`::`*`name`*`<i>`

and note after the list that *nested-name-specifier* here must contain `T`, `TT`, and/or `i`.

Add a new paragraph before paragraph 9

> If `P` has a form that contains `::`*`name`*, then `A` is a nested class or instantiation of a member class template named *name**. The types denoted by the *nested-name-specifier*s, and in the latter case the *unqualified-id*s, in `P` and `A` are compared.
>
> *Or else it would be a non-deduced context, per paragraph 5.1.

Add to subsection C.4.6:

> 14.8.2.5
> **Change:** Allowance to deduce a nested class type.
> **Rationale:** Allows use of nested classes with partial template specialization and conversion function templates.
> **Effect on original feature:** Valid C++ 2014 code may fail to compile or produce different results in this International Standard:

```
struct Base {};
template<class T> struct A {
  struct B : Base {};
};
```

```
void f(Base&);
template<class T> void f(typename A<T>::B&);  // previously could not be deduced
void g() {
  A<int>::B b;
  f(b);        // calls f(A<int>::B&); previously called f(Base&)
}
```