What Are You, Anyway?

Even experienced C++ programmers are often put off by the rather complex syntax required to program with templates. Of all the syntactic gyrations one has to undertake, none is more initially confusing than the occasional need to help the compiler disambiguate a parse.

Types of Names, Names of Types

As an example, let's look at a portion of an implementation of a simple, non-standard container template.

```
template <typename T>
class PtrList {
  public:
    //…
    typedef T *ElemT;
    void insert( ElemT );
  private:
    //…
};
```

It's common practice for class templates to embed information about themselves as nested typenames.[1] This allows us to access information about the instantiated template through the appropriate nested name.

```
typedef PtrList<State> StateList;
//…
StateList::ElemT currentState = 0;
```

The nested name `ElemT` allows us easy access to what the `PtrList` template considers to be its element type. Even though we instantiated `PtrList` with the type name `State`, the element type is `State *`. In other circumstances, `PtrList` could be implemented with a smart pointer element type, or a very sophisticated implementation of `PtrList` might vary its implementation based on the properties of the type used to instantiate it. Use of the nested type name helps to insulate users of `PtrList` from these internal implementation decisions.

Here's another non-standard container:

```
template <typename Etype>
class SCollection {
  public:
    //…
    typedef Etype ElemT;
    void insert( const Etype & );
```

[1] We'll have much more to say about this in a future Tutorial Corner, as well as mechanisms for accessing such information in a flexible and extensible way.

```
  private:
    //…
};
```

It appears that `SCollection` is designed according to the same set of naming standards as `PtrList`, in that it also defines a nested `ElemT` type name. Adherence to an established convention is useful, because (among other advantages) it allows us to write generic algorithms that work with a range of different container types. For example, we could write a simple utility algorithm that fills a conforming container with the content of an array of the appropriate element type:

```
template <class Cont>
void fill( Cont &c, Cont::ElemT a[], int len ) { // error!
    for( int i = 0; i < len; ++i )
        c.insert( a[i] );
}
```

Clueless Compilers

Unfortunately, we have a syntax error. The nested name `Cont::ElemT` is not recognized as a type name! The trouble is that, in the context of the `fill` template, the compiler does not have enough information to determine whether the nested name `ElemT` is a type name or a non-type name. The standard says that in such situations, the nested name is assumed to be a non-type name.

If at first this makes no sense to you, you're not alone. However, let's see what information is available to the compiler in different contexts. First, let's consider the situation in which we have a non-template class:

```
class MyContainer {
  public:
    typedef State ElemT;
    //…
};
//…
MyContainer::ElemT *anElemPtr = 0;
```

There's clearly no problem here, since the compiler can examine the content of the `MyContainer` class, verify that it has a member named `ElemT`, and note that `MyContainer::ElemT` is indeed a type name. Things are just as simple for a class that is generated from a class template.

```
typedef PtrList<State> StateList;
//…
StateList::ElemT aState = 0;
PtrList<State>::ElemT anotherState = 0;
```

To the compiler, an instantiated class template is just a class, and there is no difference in the access of a nested name from the class type `PtrList<State>` than there is from `MyContainer`. In either case, the compiler just examines the content of the class to determine whether `ElemT` is a type name or not.

However, once we enter the context of a template, things are different because there is less precise information available. Consider the following fragment:

```
template <typename T>
void aFuncTemplate( T &arg ) {
    …T::ElemT…
```

When the compiler encounters the qualified name `T::ElemT`, what does it know? From the template header it knows that `T` is a type name of some sort. It can also guess that `T` is a class name because we've employed the scope operator (`::`) to access a nested name of `T`. But that's all the compiler knows, because there is no information available about the content of `T`. For instance, we could call `aFuncTemplate` with a `PtrList`, in which case `T::ElemT` would be a type name:

```
PtrList<State> states;
//…
aFuncTemplate( states ); // T::ElemT is PtrList<State>::ElemT
```

But suppose we were to instantiate `aFuncTemplate` with a different type?

```
struct X {
    double ElemT;
    //…
};
X anX;
//…
aFuncTemplate( anX ); // T::ElemT is X::ElemT
```

In this case, `T::ElemT` is the name of a data member; a non-type name. What's a compiler to do? The standard tossed a coin, and in cases where it can't determine the type of a nested name, the compiler will assume the nested name is a non-type name. That is the cause of the syntax error in the `fill` function template above.

Clue In the Compiler

To deal with this situation, we must sometimes explicitly inform the compiler when a nested name is a type name.

```
template <typename T>
void aFuncTemplate( T &arg ) {
    …typename T::ElemT…
```

Here we've used the `typename` keyword to tell the compiler explicitly that the following qualified name is a type name. This allows the compiler to parse the template correctly. Note that we are telling the compiler that `ElemT` is a type name, not `T`. The compiler can already determine that `T` is a type name. Similarly, if we were to write

```
typename A::B::C::D::E
```

we'd be telling the compiler that `E` is a type name.

Of course, if `aFuncTemplate` is instantiated with a type that does not satisfy the requirements of the parsed template, it will result in a compile time error.

```
struct Z {
    // no member named ElemT…
};
Z aZ;
//…
aFuncTemplate( aZ ); // error! no member Z::ElemT
aFuncTemplate( anX ); // error! X::ElemT is not a type name
aFuncTemplate( states ); // OK. PtrList<State>::ElemT is a type name
```

Now we can rewrite the `fill` function template to parse correctly:

```
template <class Cont>
void fill( Cont &c, typename Cont::ElemT a[], int len ) { // OK
    for( int i = 0; i < len; ++i )
        c.insert( a[i] );
}
```

Gotcha: Failure to Employ `typename` with Permissive Compilers

Note that, while the use of `typename` is required to recognize a nested type name if the compiler doesn't have enough information, use of `typename` outside of a template is illegal.[2]

```
PtrList<State>::ElemT elem; // OK
typename PtrList<State>::ElemT elem; // error!
```

This is a frequent source of errors when moving code from a "template" context to a non-template context and vice versa. For example, consider the use of a template that selects one of two types at compile time based on a Boolean value:[3]

```
void f() {
    Select<cond,int,int *>::R r1; // OK
    typename Select<cond,int,int *>::R r2; // error!
    //…
}
```

Since the compiler has all the information about the template arguments available, there is no need for, and it would be illegal to employ `typename` before the `Select`. If the function `f` is rewritten as a template, however, we may use `typename` even if it is not required.

```
template <typename T>
void f() {
    Select<cond,int,int *>::R r1; // #1: OK, typename not required
    typename Select<cond,int,int *>::R r2; // #2: superfluous
    Select<cond,T,T *>::R r3; // #3: error! need typename
    typename Select<cond,T,T *>::R r4; // #4: OK
```

[2] There is talk in the standards committee of easing this somewhat.

[3] This is Andrei Alexandrescu's `Select` template. See *Modern C++ Design*.

```
    //…
}
```

Note in case #2 above that `typename` is not required, but is permitted.

The most problematic case is case #3, because many compilers will not diagnose the error, and will interpret the nested name `R` as a type name. (Yes, it *is* a type name, but it's not *supposed* to be parsed as a type name.) Later, when the code is ported to a conforming compiler the error will be diagnosed. For this reason, when programming with C++ templates, even if you must use a non-conforming compiler, it's often a good idea to check your code with at least one highly conforming compiler.

Template Names in Templates

This parsing problem is not limited to nested type names, and we encounter a similar situation with nested template names. Recall that a class or class template may have a member that is a class or function template.

For example, in an earlier installment we discussed a protopattern that we called Expanding Monostate.[4] The details of the implementation are not important, but note the use of a template member function in the `NamedMonostate` class:

```
template <typename T, int n>
struct Name {
    typedef T Type;
};

class NamedMonostate {
  public:
    template <class N>
    typename N::Type &get() { // template member function
        static typename N::Type member;
        return member;
    }
};
```

The template member function is instantiated as needed:

```
typedef Name<int,86> grossAmount;
typedef Name<double,007> percentage;
NamedMonostate nm1, nm2;
nm1.get<grossAmount>() = 12;
nm2.get<percentage>( ) = nm1.get<grossAmount>( ) + 12.2;
cout << nm1.get<grossAmount>() * nm2.get<percentage>() << endl;
```

In the code above, the compiler encounters no difficulty in determining that `get` is the name of a template. The objects `nm1` and `nm2` are of type `NamedMonostate`, and the

[4] See "Once, Weakly" for 6 November 2002.

compiler has only to look up the name `get` in the class. However, consider writing a generic function that could be used to populate an Expanding Monostate object.[5]

```
template <typename M>
void populate() {
    M m;
    m.get<grossAmount>(); // syntax error!
    M *mp = &m;
    mp->get<percentage>(); // syntax error!
}
```

Once again, the problem is that the compiler knows nothing about the name `M` except that it is a type name. In particular, because it has no information about the member name `get` of `M`, it must assume that it is a non-type, non-template name. Therefore, the angle brackets in the expression `m.get<grossAmount>()` are parsed as less-than and greater-than operators, rather than as a template argument list.

The solution is to tell the compiler that the name `get` is a template name rather than some other kind of name.

```
template <typename M>
void populate() {
    M m;
    m.template get<grossAmount>(); // OK
    M *mp = &m;
    mp->template get<percentage>(); // OK
}
```

Hideous, isn't it? Similar to the analogous use of `typename`, this particular use of the `template` keyword is only necessary and legal within a template.

Hints For Rebinding Allocators

We can also encounter the same parsing problem with a nested class template. The canonical example is in the implementation of an STL allocator.[6]

```
template <class T>
class AnAlloc {
  public:
    //…
```

[5] In point of fact, you probably wouldn't want to do this, although `populate` is actually quite an interesting function template from a philosophical point of view. It is created to force a number of template instantiations, which is a compile time operation. So there really is no need to actually call the function at runtime. However, if the function is not called, it will not be instantiated and the instantiations it provokes will not be accomplished. One alternative might be to take the function's address, rather than call it, or perform an explicit instantiation of it.

[6] If you're not familiar with STL allocators, don't worry, be happy. Previous familiarity with them is not necessary for following this discussion. An allocator is a class type that is used to customize memory management operations for STL containers. Allocators are typically implemented as class templates.

```
        template <class Other>
        class rebind {
          public:
            typedef AnAlloc<Other> other;
        };

        //…
    };
```

The class template `AnAlloc` contains the nested name `rebind`, which is itself a class template. It is used within the STL framework to create allocators "just like" the allocator that was used to instantiate a container, but for a different element type. For example:

```
typedef AnAlloc<int> AI; // original allocator allocates ints
typedef AI::rebind<double>::other AD; // new one allocates doubles
typedef AnAlloc<double> AD; // legal! this is the same type
```

It may look a little odd, but using the `rebind` mechanism allows one to create a version of an existing allocator for a different element type without knowing the type of the allocator or the type of the element.

```
typedef SomeAlloc::rebind<ListNode>::other NewAlloc;
```

If the type name `SomeAlloc` follows the STL convention for allocators, then it will have a nested `rebind` class template. Essentially, we've said, "I don't know what kind of allocator this type is, and I don't know what it allocates, but I want an allocator just like it that allocates `ListNode`s!

This level of ignorance usually occurs only within a template, where precise types and values are not known until much later, when the template is instantiated. Consider the implementation of an STL-compliant list container of some sort. The list template takes two template arguments; an element type (`T`), and an allocator type (`A`) that can allocate elements. (Like the standard containers, our list provides a default allocator argument.)

```
template < typename T, typename A = std::allocator<T> >
class OurList {
    template <typename S>
    struct Node {
        //…
    };
    typedef A::rebind< Node<T> >::other NodeAlloc; // error!
};
```

As is typical for lists and other node-based containers, our list does not actually allocate and manipulate `T`s. Rather, it allocates and manipulates nodes that contain a member of type `T`. This is the situation we described earlier. We have some sort of allocator that knows how to allocate objects of type `T`, but we want to allocate objects of type `Node<T>`. However, when we attempt to `rebind`, we get a syntax error.

Once again, the problem is that the compiler has no information about the type name `A` at this point other than that it is a type name. The compiler has to make the assumption that the nested name `rebind` is a non-template name, and the angle bracket that follows is parsed as a less-than. But our troubles are just beginning. Even if the compiler were able to determine that `rebind` is a template name, when it reached the (doubly) nested name `other`, it would have to assume that it's a non-type name! Time for some clarification. The typedef must be written as follows:

```
typedef typename A::template rebind< Node<T> >::other NodeAlloc;
```

The use of `template` tells the compiler that `rebind` is a template name, and the use of `typename` tells the compiler that the whole mess refers to a type name. Simple, right?