A Description of the C++ `typename` keyword

The purpose of this document is to describe the reasoning behind the inclusion of the `typename` keyword in standard C++, and explain where, when, and how it can and can't be used.

Note: This page is correct (AFAIK) for C++98/03. The rules have been loosened in C++11.

Table of contents

A secondary use

There is a use of `typename` that is entirely distinct from the main focus of this discussion. I will present it first because it is easy. It seems to me that someone said "hey, since we're adding `typename` anyway, why not make it do this" and people said "that's a good idea."

Most older C++ books, when discussing templates, use syntax such as the following:

```
template <class T> ...
```

I know when I was starting to learn templates, at first I was a little thrown by the fact that `T` was prefaced by `class`, and yet it was possible to instantiate that template with primitive types such as `int`. The confusion was very short-lived, but the use of `class` in that context never seemed to fit entirely right. Fortunately for my sensibilities, it is also possible to use `typename`:

```
template <typename T> ...
```

This means *exactly the same thing* as the previous instance. The `typename` and `class` keywords can be used interchangeably to state that a template parameter is a type variable (as opposed to a non-type template parameter).

I personally like to use `typename` in this context because I think it's ever-so-slightly clearer. And maybe not so much "clearer" as just conceptually nicer. (I think that good names for things are very important.) Some C++ programmers share my view, and use `typename` for templates. (However, later we will see how it's possible that this decision can hurt readibility.) Some programmers make a distinction between templates that are fully generic (such as the STL containers) and more special purpose ones that can only take certain classes, and use `typename` for the former category and `class` for the latter. Others use `class` exclusively. This is just a style choice.

However, while I use `typename` in real code, I will stick to `class` in this document to reduce confusion with the other use of `typename`.

The *real* reason for `typename`

This discussion I think follows fairly closely appendix B from the book *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond* by David Abrahams and Aleksey Gurtovoy, though I don't have it in front of me now. If there are any deficiencies in my discussion of the issues, that book contains the clearest description of them that I've seen.

Some definitions

There are two key concepts needed to understand the description of `typename`, and they are *qualified* and *dependent* names.

Qualified and unqualified names

A qualified name is one that specifies a scope. For instance, in the following C++ program, the references to `cout` and `endl` are qualified names:

```
#include <iostream>

int main()  {
   std::cout << "Hello world!" << std::endl;
}
```

In both cases, the use of `cout` and `endl` began with `std::`.

Had I decided to bring `cout` and `endl` into scope with a `using` declaration or directive*, and used just "`cout`" by itself, they would have been unqualified names, because they would lack the `std::`.

(* Remember, a using declaration is like `using std::cout;`, and actually introduces the name `cout` into the scope that the `using` appears in. A using directive is of the form `using namespace std;` and makes names visible but doesn't introduce anything. [12/23/07 -- I'm not sure this is true. Just a warning.])

Note, however, that if I had brought them into scope with `using` but still used `std::cout`, it remains a qualified name. The qualified-ness of a name has nothing to do with what scope it's used in, what names are visible at that point of the program etc.; it is solely a statement about the name that was used to reference the entity in question. (Also note that there's nothing special about `std`, or indeed about namespaces at all. `vector<int>::iterator` is a nested name as well.)

Dependent and non-dependent names

A dependent name is a name that depends on a template parameter. Suppose we have the following declaration (not legal C++):

```
template <class T>
class MyClass {
   int i;
   vector<int> vi;
   vector<int>::iterator vitr;

   T t;
   vector<T> vt;
   vector<T>::iterator viter;
};
```

The types of the first three declarations are known at the time of the template declaration. However, the types of the second set of three declarations are *not* known until the point of instantiation, because they depend on the template parameter `T`.

The names `T`, `vector<T>`, and `vector<T>::iterator` are called dependent names, and the types they name are dependent types. The names used in the first three declarations are called non-dependent names, at the types are non-dependent types.

The final complication in what's considered dependent is that typedefs transfer the quality of being dependent. For instance:

```
typedef T another_name_for_T;
```

`another_name_for_T` is still considered a dependent name despite the type variable `T` from the template declaration not appearing.

Note: If you're know some advanced type theory, note that C++'s notion of a dependent name has almost nothing to do with type theorists' dependent types.

Some other issues of wording

Note that while there is a notion of a dependent type, there is not a notion of a qualified type. A type can be unqualified in one instance, and qualified the next; the qualification is a property of a particular naming of a type, not of the type itself. (Indeed, when a type is first defined, it is always unqualified.)

However, it will be useful to refer to a qualified type; what I mean by this is a qualified name that refers to a type. I will switch back to the more precise wording when I talk about the rules of `typename`.

The problem

So now we can consider the following example:

```
template <class T>
void foo() {
   T::iterator * iter;
   ...
}
```

What did the programmer intend this bit of code to do? Probably, what the programmer intended was for there to be a class that defined a nested type called `iterator`:

```
class ContainsAType {
   class iterator { ... }:
   ...
};
```

and for `foo` to be called with an instantiation of `T` being that type:

```
foo<ContainsAType>();
```

In that case, then line 3 would be a declaration of a variable called `iter` that would be a pointer to an object of type `T::iterator` (in the case of `ContainsAType`, `int*`, making `iter` a double-indirection pointer to an int). So far so good.

However, what the programmer didn't expect is for someone else to come up and declare the following class:

```
class ContainsAValue {
   static int iterator;
};
```

and call `foo` instantiated with it:

```
foo<ContainsAValue>();
```

In this case, line 3 becomes a statement that evaluates an expression which is the product of two things: a variable called `iter` (which may be undeclared or may be a name of a global) and the static variable `T::iterator`.

Uh oh! The same series of tokens can be parsed in two entirely different ways, and there's no way to disambiguate them until instantiation. C++ frowns on this situation. Rather than delaying interpretation of the tokens until instantiation, they change the languge:

Before a qualified dependent type, you need `typename`

To be legal, assuming the programmer intended line 3 as a declaration, they would have to write

```
template <class T>
void foo() {
   typename T::iterator * iter;
   ...
}
```

Without `typename`, there is a C++ parsing rule that says that qualified dependent names should be parsed as non-types even if it leads to a syntax error. Thus if there was a variable called `iter` in scope, the example would be legal; it would just be interpreted as multiplication. Then when the programmer instantiated `foo` with `ContainsAType`, there would be an error because you can't multiply something by a type.

`typename` states that the name that follows should be treated as a type. Otherwise, names are interpreted to refer to non-types.

This rule even holds if it doesn't make sense **even if it doesn't make sense to refer to a non-type**. For instance, suppose we were to do something more typical and declare an iterator instead of a pointer to an iterator:

```
template <class T>
void foo() {
   typename T::iterator iter;
   ...
}
```

Even in this case, `typename` is required, and omitting it will cause compile error. As another example, typedefs also require use:

```
template <class T>
void foo() {
   typedef typename T::iterator iterator_type;
   ...
}
```

The rules

Here, in excruciating detail, are the rules for the use of `typename`. Unfortunately, due to something which is hopefully not-contagious apparently affecting the standards committee, they are pretty complicated.

- `typename` is *prohibited* in each of the following scenarios:
  - Outside of a template definition. (Be aware: an explicit template specialization (more commonly called a *total* specialization, to contrast with partial specializations) is not itself a template, because there are no missing template parameters! Thus `typename` is always prohibited in a total specialization.)
  - Before an unqualified type, like `int` or `my_thingy_t`.
  - When naming a base class. For example, `template <class C> class my_class : C::some_base_type { ... };` may *not* have a `typename` before `C::some_base_type`.
  - In a constructor initialization list.
- `typename` is *mandatory* before a qualified, dependent name which refers to a type (unless that name is naming a base class, or in an initialization list).
- `typename` is *optional* in other scenarios. (In other words, it is optional before a qualified but *non*-dependent name used within a template, except again when naming a base class or in an initialization list.)

Again, these rules are for standard C++98/03. C++11 loosens the restrictions. I will update this page after I figure out what they are.