Down with **typename**!

If \( X<T>::Y \) — where \( T \) is a template parameter — is to denote a type, it must be preceded by the keyword `typename`; otherwise, it is assumed to denote a name producing an expression. There are currently two notable exceptions to this rule: `base-specifiers` and `mem-initializer-ids`. For example:

```cpp
template<class T> struct D: T::B { // No `typename` required here
  
};
```

Clearly, no `typename` is needed for this `base-specifier` because nothing but a type is possible in that context. However, there are several other places where we know only a type is possible and asking programmers to nonetheless specify the `typename` keyword feels like a waste of source code space (and is detrimental to readability).

I therefore propose we make `typename` optional in the following places:

- The top-level `decl-specifier-seq` of a `simple-declaration` in namespace scope.
- The top-level `decl-specifier-seq` of a `member-declaration` (in class scope).
- The top-level `decl-specifier-seq` of a `parameter-declaration` in a class or namespace scope, or in a lambda.
- A `trailing-return-type`.
- The `defining-type-id` of an alias declaration.
- The `type-id` of a `static_cast`, `const_cast`, `reinterpret_cast`, or `dynamic_cast`.
- The default argument of a `type-parameter` of a template.
- The `type-id` or `new-type-id` or a `new-expression`. 
With the changes above, we’d be able — for example — to write

```cpp
template<class T> T::R f(T::P);
template<class T> struct S {
    using Ptr = PtrTraits<T>::Ptr;
    T::R f(T::P p) {
        return static_cast<T::R>(p);
    }
    auto g() -> S<T*>::Ptr;
};
```

instead of the currently-required:

```cpp
template<class T> typename T::R f(typename T::P);
template<class T> struct S {
    using Ptr = typename PtrTraits<T>::Ptr;
    typename T::R f(typename T::P p) {
        return static_cast<typename T::R>(p);
    }
    auto g() -> typename S<T*>::Ptr;
};
```

A cursory read through some common standard library headers suggests that by-far most occurrences of `typename` for the purpose of disambiguating type names from other names can be eliminated with these new rules.

The EDG front end has an `implicit typename` mode to emulate pre-C++98 compilers that didn’t parse templates in their generic form. Although that mode doesn’t exactly cover the contexts where I’m proposing to make `typename` optional, the implementation effort is similar (and not excessively expensive).