Doc No: SC22/WG21/1420
J16/03-0002

Date: March 3, 2003

Project: JTC1.22.32

Reply to: Carl Daniel
Email: cpdaniel@mvps.org

Proposed Addition to C++: Class Namespaces

1. The Problem

C++ classes have always been closed, such that once a class definition is ended, it cannot be opened again, and nothing can be added to it. We propose allowing classes to be reopened for the purpose of defining member functions, static data members, and types already declared in the class definition. We call this reopened class a "class namespace".

The present situation, and the proposed solution is particularly significant for developers of template classes, who now have to repeat a sometimes large and distracting template declaration before each and every out-of-line member definition. This situation creates a “Catch-22” for the template implementer: to move member definitions out of line means a significant amount of extra work, especially when template parameters change, but to leave member definitions inline may cause unnecessary increases in code size, due to the implicit inlining of those members.

1.1 Motivating Analogy

When defining a class in C++, the programmer has the option to place the definition of member functions inline in the class, or to place those definitions out of line, or to use a combination of the two techniques.

```
class a_class
{
public:
      // declaration, no definition
      void f();

      // inline definition
      void g()
      {
      }
};

// out of line definition for a_class::f
void a_class::f()
{
}
```

With simple classes, the out-of-line syntax is straightforward, if slightly repetitive. With template classes, and especially with nested template classes, the visual noise introduced by re-specifying the class name can become both distracting and an impediment to productivity, especially when the template arguments must change.

```
template <class T> class outer
{
      template <class U> class inner
      {
            // declaration only, no definition
            T f(U&);
      };
};

// out of line definition
template <class T>
template <class U>
T outer<T>::inner<U>::f(U&)
{
}
```

As the number of template functions or template parameters increases, the overhead of (and chance for error in) repeating the class specifier for each function increases. For a template with several parameters, it's not uncommon for the class specifier to be larger than the bodies of many of the member functions. Further, the need to repeat the template formal parameters with each function opens the door for non-uniformity of implementation, and higher maintenance costs.

In contrast to class member functions, free-functions declared in namespaces incur no such textual overhead, since namespaces can be re-opened at will, while classes are closed.

```
// in a header file
namespace outer
{
      namespace inner
      {
            // declaration only, no definition
            void f();
            void g();
      }
}

// elsewhere – e.g. in an implementation file
namespace outer
{
      namespace inner
      {
            // definitions
            void f()
            {
            }

            void g()
            {
            }
      }
}
```

The ability to re-open a namespace is essential to allow members of a namespace (such as `std`) to be specified in several header files. As a side-effect, the fact that namespaces can be re-opened saves the programmer the need to use a fully-qualified name when defining a free-function that was declared in a namespace. The lack of similar freedom for class member implementations is the problem at hand.

A solution to this problem would support several of the committee's goals for improvements in the C++ language: it would improve support for library building, since library authors could concentrate more on the meat of their library, and less on the mundane syntax details; it would improve support for generic programming, by providing a kind of "generic definition space", which complements the "generic declaration space" which is already available in the language; it would make C++ easier to learn and understand, by simplifying the definition of out-of-line class members.

2. The Proposal

The C++ standard (§7.3/1) defines a namespace as "an optionally named declarative region". This proposal describes "class namespaces". A class namespace is a "named definitive region". The name of a class namespace name is always identical to the name of an existing class definition. Unlike an ordinary namespace, a class namespace may contain only definitions, no declarations.

The proposal is to allow the "namespace" (or scope) of a class to be re-opened for the purpose of adding definitions corresponding to existing declarations. The goal is that given:

```
class A
{
      return-type f(parameter-list);
};
```

that

```
namespace class A
{
      return-type f(parameter-list)
      {
            // statements
      }
}
```

means exactly the same thing as

```
return-type A::f(parameter-list)
{
      // statements
}
```

with constructs involving static members (both functions and data) and nested types behaving in an analogous manner.

2.1 Simple classes

The proposed extension allows "class namespaces" to be re-opened using a "class namespace definition". The proposed syntax follows:

```
// class definition
class A
{
      // declarations
};

// elsewhere
// class namespace definition
namespace class A
{
      // definitions
}
```

Within a re-opened "class namespace":

- no new declarations may appear – only definitions of previously declared members are allowed.
- Definitions may include member functions, static data members, and member types.
- Member functions defined within the re-opened namespace are not implicitly inline (in contrast to members defined in the original class declaration).
- Member function definitions may include the `inline` modifier to designate a member function as inline. No access specifiers may appear within a class namespace definition.
- No virtual or static modifiers may appear within a class namespace definition.

2.2 Template classes

The "class namespace" of a class template may be re-opened:

```
template <class T> class A
{
      // declarations
};

// elsewhere
template <class T>
namespace class A
{
      // definitions
}
```

2.3 Specialization

The "class namespace" of a class template specialization (full or partial) may be re-opened:

```
// full specialization
template <> class A<void*>
{
      // declarations
};

// partial specialization
template <class T> class A<T*>
{
      // declarations
};

// elsewhere

// add definitions to a full specialization
template<>
namespace class A<void*>
{
      // definitions
}

// add definitions to a partial specialization
template <class T>
namespace class A<T*>
{
      // definitions
}
```

2.5 Nested classes

The "class namespace" of a nested class may be re-opened:

```
template <class T> class outer
{
      template <class U> class inner
      {
            // declarations
      };
};

// elsewhere
template <class T>
namespace class outer
{
      template <class U>
      namespace class inner
      {
            // definitions
      }
}

// equivalent
template <class T>
template <class U>
namespace class outer<T>::inner
{
      // definitions
}
```

The nested class namespace must match a class previously declared in the enclosing class.

2.6 Incomplete nested classes

The definition of an incomplete nested class may be given inside a class namespace.

```
class outer
{
      class inner;
};

namespace class outer
{
      class inner
      {
            // declarations and definitions
      };
}

namespace class outer::inner
{
      // definitions
}
```

Note that when an incomplete nested class is defined within a class definition namespace that declarations may appear within that definition. These declarations, of course, belong to the namespace of the inner class (that's being defined) and not to the re-opened class definition namespace of the outer class. From the viewpoint of the re-opened class, only definitions appear.

2.7 Legal Scopes

A class namespace definition may appear in the same scopes that a namespace definition may appear: in namespace scope or in global scope. A class namespace definition may also appear within another class namespace definition.

2.8 Applicable Types

The following types may be "re-opened" using a class namespace definition:

- A complete class type
- A complete class template

Example:

```
struct X
{
      struct Y;
};

// legal
namespace struct X
{
      // illegal – Y is incomplete
      namespace struct Y
      {
      }
}

// illegal – Y is incomplete
namespace struct X::Y
{
}

struct X::Y
{
};

// legal – X::Y is now complete
namespace struct X::Y
{
}

namespace struct X
{
      // legal – X::Y is now complete
      namespace struct Y
      {
      }
}
```

3. Interactions and Implementability

3.1 Interactions

The proposed feature is intended to be a natural simplification of existing C++ syntax – one whose meaning is readily apparent to both new and experienced C++ programmers. Interactions with the rest of the language are minimal, as the proposal adds new syntax only – no new types are created, and no new semantics are attached to any existing language construct.

The proposed feature does not interact with legacy code at all, since no well-formed program can contain the token sequence `namespace class` under the current language rules.

This is a generally useful extension, particularly for authors and maintainers of template libraries. Under the proposal, a parity between class definitions and the definitions of their member functions is created. Rather than repeatedly specifying the class when defining member functions, a single specification will suffice for any sized group of member definitions.

3.2 Implementability

No sample implementation is available at this time, but it's not anticipated that the proposal should be difficult to implement. The proposal does not introduce new lexical elements, and it combines existing tokens in ways that are always erroneous under the current language rules. A proof-of-concept could be implemented as a standalone translator – accepting the proposed syntax and emitting ISO 14882:1998(E) compliant syntax.

3.3 Alternative Designs

Some participants in the newsgroup thread that led to this proposal suggested that instead of overloading the `namespace` keyword, that a new keyword, such as `implement` be added instead.

This proposal favors the use of the namespace keyword for two reasons:

1. It avoids adding another keyword.
2. The proposed extension is semantically very close to the current usage of the `namespace` keyword and should be easy to grasp by existing and new C++ programmers.

Some readers of this proposal question whether it's necessary to prohibit all forms of declarations within a class namespace. This proposal favors prohibiting all declarations since doing so retains the meaning of a class definition as it exists in the current language. However, arguments can be made for allowing certain kinds of declarations to appear within a re-opened class namespace:

- Typedefs.
- Private, non-virtual functions.
- Private, static functions.
- Previously undeclared overrides of inherited virtual functions.

If any declarations were allowed in a re-opened class namespace, the visibility of those declarations would need to be specified. For example:

```
class A
{
};

namespace class A
{
      // public? private? protected?
      typedef int size_type;
}

A::size_type s;   // legal?
```

The least troublesome interpretation would be that declarations added to a class within a class namespace are visible only in the block in which they appear. Doing otherwise would require that public/private/protected modifiers be allowed within the class namespace block, and would create a situation where the interfaces exposed by a class differ depending on the point of use. A class defined in a header file, and implemented using class namespaces in two separate translation units would have 3 different interfaces: that defined by the class, that defined by the class namespace in one translation unit, and that defined by the class namespace in the other translation unit. This is A Bad Thing.

4. Acknowledgements

The ideas expressed in this proposal originated in postings on `comp.std.c++` by Thorsten Ottosen (message ID `b03ef0$i5n$1@sunsite.dk`).

The particular syntax recommended in this document was first suggested by Holger Grund, in private correspondence.

Bill Clark, Doug Harrison, Tomas Restrepo, Tom Serface, Brandon Bray, Jeff Peil, Chris Lucas and Mark Hall provided valuable feedback on early drafts of this proposal.