x3j16/93-0055, WG21/N0262

Name Space Management in C++ (revised)

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This is a proposal for a mechanism for defining name spaces in such a way that users can compose programs from separately developed fragments (libraries) without worrying too much about names used for different classes or functions in different fragments. The scheme is based on `namespace` declarations for gathering otherwise global declarations into a separate name space, qualification for explicit access to a namespace, and `using` declarations for implicit access to a namespace. Two alternative designs, one without `using` and one where a namespace is a kind of class, are discussed in appendices. Similarly, the difficult design decision for overloading across namespaces and for name injection are discussed in appendices.

1 The Problem

Like C, C++ provides a single global name space into which every name that doesn't conveniently fit into a class or a function must be entered. This makes it (unnecessarily) difficult to write program fragments that can be linked together without fear of name clashes. For example:

```
// my.h:
        char f(char);
        int f(int);
        class String { /* ... */ };

// your.h:
        char f(char);
        double f(double);
        class String { /* ... */ };
```

Given these definitions, a third party cannot easily use both `my.h` and `your.h`.

Note that some of these names will appear in object code and that some programs will be shipped without source. This implies that ‘‘macro-like’’ schemes that change the appearance of programs without actually changing the names presented to a linker are insufficient. Further, I assume that there are too many linkers and too many object code formats around to make it feasible to change them. For a solution to be useful for us it must require only facilities provided by almost all current linkers. This implies that disambiguation must be compiled into the object code by providers of program fragments. In particular, library providers will have to use a technique that allows users to disambiguate. I foresee no problems getting library providers to cooperate – that is, to use a name space resolution scheme – because they (partly through their users) are the main sufferers in the current situation.

Workarounds

There are several workarounds. For example:

```
// my.h:
        char  my_f(char);
        int my_f(int);
        class my_String { /* ... */ };
```

```
// your.h:
        char yo_f(char);
        double yo_f(double);
        class yo_String { /* ... */ };
```

This approach is not uncommon, but it is quite ugly and – unless the prefix strings are short – quite unpleasant for the user. Macro hackery can make this approach even nastier (or even nicer, if you happen to like macros):

```
// my.h:
        #define my(X) myprefix_##X

        char  my(f)(char);
        int my(f)(int);
        class my(String) { /* ... */ };

// your.h:
        #define yo(X) your_##X

        char yo(f)(char);
        double yo(f)(double);
        class yo(String) { /* ... */ };
```

The idea is to allow longer prefixes in the real name used for linkage while leaving the names used in the program short. As with all macro schemes, this creates a problem for tools: Either the tool keeps track of the mapping (complicating the tool) or the user will have to do so (complicating programming and maintenance).

An alternative approach – often preferred by people who dislike macros – is to wrap related information into a class:

```
// my.h:
        class My {
        public:
                static char f(char);
                static int f(int);
                class String { /* ... */ };
        };

// your.h:
        class Your {
        public:
                static char f(char);
                static double f(double);
                class String { /* ... */ };
        };
```

Unfortunately, this approach suffers from many little inconveniences. Not all global declarations can be simply transferred into a class and some change their meaning if you do so. For example, global functions and variables must be specified as `static` members to avoid semantic changes and the function bodies and initializers must in general be separated from their declarations:

```
int f(int) { /* ... */ }
char a[] = "asdfg";
```

cannot become simply:

```
class Nice_try {
        int f(int) { /* ... */ }
        char a[] = "asdfg";
};
```

but must be reworked into:

```
class better_but_not_nice {
public:
        static int f(int);
        static char a[];
};

int better_but_not_nice::f(int) { /* ... */ }
char better_but_not_nice::a[] = "asdfg";
```

Furthermore, there is no way of making use of these ‘‘wrapped’’ declarations as notationally conveniently as use of ordinary globals. For example:

```
void h()
{
        int a = f(2);
        String s = "asdf"
}
```

looks noticeably cleaner than

```
void h()
{
        int a = my_f(2);
        my_String s = "asdf"
}
```

and

```
void h()
{
        int a = My::f(2);
        My::String s = "asdf"
}
```

especially when the prefix has to be repeated hundreds of times in a source file. There is no way of saying ‘‘in this program I use the names from the `My` library.’’ For a further discussion of classes and namespaces, see Appendix B.

Ideals for a solution

There are many mechanisms that can be used to provide solutions to namespace problems. Indeed most languages can claim to have at least the rudiments of one. For example, C has its static functions, Pascal its nested scopes, C++ its classes, but we need to go to languages such as PL/I, Ada, Modula-2, Modula-3, ML, and CLOS for more complete solutions.

So what would a good namespace mechanism give us in C++? A lengthy and voluminous discussion on the x3j16 extensions working group mailing list provided a list:

[1] The ability to link two libraries without name clashes.

[2] The ability to introduce names without fear of clashing with someone else’s names (e.g. names used in a library I haven’t heard of, or names I haven’t heard of in a library I thought I knew).

[3] The ability to add a name to the implementation of a library without affecting its users.

[4] The ability to select names from two different libraries even if those two libraries use the same names.

[5] The ability to resolve name clashes without modifying source code statements (i.e. through declarations manipulating the name space resolution).

[6] The ability to add a name to a namespace without fear of causing a quiet change to code using other namespaces (we cannot provide such a guarantee for code using the namespace being added to).

[7] The ability to avoid clashes among name space names (in particular, the ability to have the ‘‘real’’ or linkage name longer than the name used in user code).

[8] The ability to use the name space mechanism to deal with the standard libraries.

[9] C and C++ compatibility.

[10] No added cost in link-time or run-time for the users of namespaces.

[11] No added verbosity for the users of namespaces compared to users of global names.

[12] The ability to indicate explicitly where a name is supposed to come from in code using the name.

In addition, a good solution must be simple. We might define ‘‘simple’’ as:

[1] A mechanism that can be explained to the degree needed for serious use in less than ten minutes. (Explaining any mechanism to the satisfaction of language lawyers will take much longer).

[2] Something a C++ compiler writer can implement in less that two weeks.

Naturally, simplicity in this sense cannot be proven rigorously. For example, the time needed to understand something will vary greatly between people with different backgrounds and different levels of ability. However, after the first version of this paper was written I have ‘‘tested’’ this proposal for simplicity according to these criteria. A pilot implementation was completed in five days, and I have explained the basics of namespaces to several people in less than ten minutes using just a couple of foils. Their follow-up questions showed understanding and the ability to deduce some of the uses of namespaces that I hadn’t explained. I am now satisfied that the proposal is ‘‘simple enough.’’

In addition, there are some properties that have been asked for but that I don’t propose to support directly with new features:

[1] The ability to take binaries with clashing link names and link them together. (This can be done by tools in all systems, but I don’t see a language mechanism that could easily be implemented without significant effort or overhead on *all* systems). See also §1 and §7.

[2] The ability to provide arbitrary synonyms for names used in libraries. (Existing mechanisms, such as typedef, references, and macros, provide mechanisms for providing synonyms in many cases). See also Appendix C.

Naturally, it is possible to add criteria to these lists and no two people will agree to the exact importance of the criteria, but these lists gives an idea of the complexity of the problem and the demands that a solution must meet.

As I work through the proposed namespace mechanism, its uses, its possible misuses, possible alternative designs, and details, you will will probably get lost at times. Let me therefore state that the proposed solution is fundamentally simple. It provides four new mechanisms:

[1] A mechanism for defining a scope that holds what have traditionally been global declarations in C and C++: a namespace. Such scopes can be named and a namespace’s members can be named using the traditional notation for class members: `namespace_name::member_name`. In fact, a class scope can be seen as a special case of a namespace scope.

[2] A mechanism for defining a local synonym for a namespace name.

[3] A mechanism to allow explicitly specified members of a namespace to be accessed without the explicit `namespace_name::` qualification: a *using-declaration*.

[4] A mechanism to allow *all* members of a namespace to be accessed without the explicit `namespace_name::` qualification: a *using-directive*.

I believe this suffices to meet the criteria above. In addition, it solves a long-standing problem with access to base class members from a derived class scope (see §10) and renders `static` redundant as used for global names (see §9).

The text suggested for addition to the reference manual (§11) is about one page and a half and makes some existing text about classes redundant.

2 Namespaces

Consider a language construct specifically providing name spaces:

```
namespace A {
        void f(int);
        void f(char);
        class String { /* ... */ };
        // ...
}
```

The names declared within the *namespace* braces are in namespace `A` and do not collide with global names or names in any other name space. The semantics of declarations (including definitions) in a name space are exactly that of global declarations except that the scope of their names are restricted to the name space.

To use a name from a namespace `A` you can either explicitly qualify it

```
              A::String s;
```

or import it into a scope

```
              using A::String;
              String s;       // meaning A::String
```

Using Declarations

A *using-declaration* `NS::m` declares a local name `m` that can be used to access whatever object, type, functions, etc., is named by `m` in `NS`. One can import either a single name or a set of names from a namespace:

```
      using X::f;            // import f from X
      using X::(f, g, h);    // import f, g, and h from X
```

See §11 for a grammar.

Redundant `using` declarations are allowed just like other redundant declarations.

The usual scope rules apply:

```
      void g(int i)
      {
              using A::String;

              String s = "asdf"; // A::string

              if (i) {
                      extern void h();
                      using A::f;

                      // we can use h() and A::f() here
              }
              else {
                      //  h() and f() are not in scope here

                      struct String // hides A::String
                              { /* ... */ };

                      String ss;      // local String
              }

              //  h() and f() are not in scope here

              String s2 = "asdf"; // A::string
      }
```

Once a name is declared locally with a *using-declaration*, ambiguity detection and overload resolution apply as usual. For example:

```
extern void f(double);
using A::f;

void hh()
{
        f(2.0);       // ::f(double)
        f(1);         // A::f(int)
        A::f(2.0);    // A::f(int)
        ::f(1);       // A::f(int)
                      //    there are three f()s declared in the
                      //    global scope (one declared with
                      //    'extern,' two with 'using') ambiguity
                      //    resolution choses the f(int)
        f('c');       // A::f(char)
}
```

Note that `using A::f` brings in all `f`s from `A`. See also §5.

It is an error to declare `X::m` unless `X` is a namespace and there is an `m` in `X`. In `X::m`, it is known that `X` must be a namespace name because it precedes `::` Thus, following the rule for class names (ARM §5.1), `X` can be found even if it has been hidden by a local non-namespace name:

```
namespace X {
        int m1;
        int m2;
}

int Y;

void f()
{
        int X;
        using X::m1;    // ok (the local X isn't a namespace name)

        int m2;
        using X::m2;    // error: two declarations of m2 in f()

        using X::m3;    // error: no m3 in X

        using Y::m;     // error: no namespace Y in scope
        using Z::m;     // error: no namespace Z in scope
}
```

Using Directives

Mentioning every name from a namespace that one wants to use explicitly in a *using-declaration* can be tedious. Long lists of names are also potentially error-prone because they tend to be incomplete and not resilient to changes in the namespace they refer to. Consequently, a mechanism is provided to make all names from a namespace available without qualification and without mention of individual names. For example:

```
void g(int i)
{
        using namespace A;
        String s = "asdf";
}
```

or equivalently

```
        using namespace A;

        void g(int i)
        {
                String s = "asdf";
        }
```

A *using-directive* makes the names from the namespace available as if they had been declared without a namespace at the point where their namespace was declared; it does *not* define local aliases for the names in the namespace. One might think as a *using-directive* as granting a key that allows a namespace to be opened when found during a name lookup. For example, a namespace

```
        namespace A {
                int a, b;
        };
```

looks like plain

```
                int a, b;
```

to code for which a

```
        using namespace A;
```

is in scope. In this, a *using-directive* differs from a *using-declaration* of specific names. For example:

```
        namespace X {
                int i, j, k;
        }

        int k;

        void f1()
        {
                int i = 0;
                using namespace X; // make names from X accessible
                i++;               // local i
                j++;               // X::j
                k++;               // error: X::k or global k ?
        }

        void f2()
        {
                int i = 0;
                using X::i;     // error: i declared twice in f2()
                using X::j;
                using X::k;     // hides global k
                i++;
                j++;            // X::j
                k++;            // X::k
        }
```

A *using-declaration* adds to a local scope. A *using-directive* does not; it simply renders names accessible.

For more name resolution details see §5. Note that the meaning of a *using-directive* doesn't depend on exactly where it is placed as long as it is in scope. For example,

```
        using namespace X;      // make names from X accessible

        void f1()
        {
                int i = 0;
                i++;            // local i
                j++;            // X::j
                k++;            // error: X::k or global k ?
        }
```

is equivalent to the definition of `f1()` above.

Having *namespace-declaration*s introduce local aliases and *namespace-directive*s not to ensures that whenever a name is explicitly named in a local declaration then that declaration determines the meaning of that name in the local context without interference from other declarations of that name in other contexts.

Originally, I thought the shorter form

```
        using X;
```

could be used instead of the more explicit

```
        using namespace X;
```

However, in real use people was confused about the difference between a *using-declaration* and a *using-directive*. Part of the reason was that there was no strong syntactic clue to the difference. The implementation also proved easier given the (logically redundant) added `namespace`.

3 How to Use Namespaces

A supplier, say a library vendor, will present an interface to a set of services in the form of a namespace. For example:

```
        namespace my_library {
                // classes
                // typedefs
                // global variable declarations
                // templates
                // global function declarations
                // consts
                // inline functions
                // etc.
        }
```

Typically, this will be placed in a header file so that a user includes the namespace like this:

```
        #include "my_library.h"

        // use my_library
```

To access the library facilities, the user has several choices. One can crudely and effectively make all the names from the library available in the global scope:

```
        #include "my_library.h"
        using namespace my_library;
```

This is equivalent to a traditional `#include` of a header file that doesn't use name spaces. After

```
        using namespace my_library;
```

every name from `my_library` is available without qualification. If used, a name from `my_library` that clash with a global name cause a compile-time error unless the clash is resolved by the function and operator overloading rules. Overload resolution applies across namespaces where *using-directive*s or *using-declaration*s have made names accessible; see Appendix D.

A more selective approach will be taken by users who worry about name clashes, about function name resolution, or about documenting which facilities from `my_library` are used. For example:

```
#include "my_library.h"
using my_library::(String, f);  // String and f from my_library
                                // can be used from here on

String s;       // my_library::String

void g()
{
        f();    // my_library::f
        // ...
}

void h()
{
        using my_library::g;  // h() can also use my_library::g

        g();    // my_library::g
        f();    // my_library::f
        // ...
}
```

Namespace Aliases

If the repetition of the namespace name gets tedious a synonym can be introduced:

```
#include "my_library.h"
namespace lib = my_library;

using lib::(String, f);  // lib::String and lib::f
                         // can be used from here on

String s;       // lib::String

// ...
```

In addition to notational convenience, the use of a synonym also makes it easier to change libraries. For example, changing the first two lines of my program (only) to:

```
#include "your_library.h"
namespace lib = your_library;
```

will ensure that all of the code uses `your_library`. We considered using `typedef` for introducing synonyms for namespaces, but a namespace isn't a type. You can introduce as many synonyms for a namespace as you like, but the real name of the namespace – as known to the linker – is still the original name (only).

A namespace alias cannot be used to add more members to a namespace:

```
namespace A {
        // ...
}

namespace B = A;

namespace A {   // ok, see §5 ‘‘dispersed namespace definitions’’
        // ...
}

namespace B {   // error: B is a namespace alias
        // ...
}
```

Suppliers
Library suppliers need to provide an implementation for the facilities offered by the interface. Typically, this will involve including the header containing the the namespace declaration (just like users do) and then use qualification in the definition of types, functions, objects, etc.:

```
#include "my_library.h"

void my_library::f()
{
        // ...
}

int my_library::a = 7;
```

Alternatively, an implementor can wrap the definitions in a namespace declaration:

```
#include "my_library.h"

namespace my_library {

        void f()        // define my_library::f()
        {
                // ...
        }

        int a = 7;      // define my_library::a

}
```

Here `f()` and `a` are defined in the scope of `my_library`.

A *using-declaration* or *using-directive* gives access to namespace names only when looking for a *use* of a name; it does not affect definitions of new types, objects, functions, etc. For example:

```
#include "my_library.h"
using my_library;

void f()        // define ::f(), not mylibrary::f()
{
        // ...
}
```

does *not* define `mylibrary::f()` but a global `f()`. This could of course be different from what the programmer expected, but that mistake will lead to an undefined `mylibrary::f()` which will eventually be detected. Also, global functions (not in namespaces) should eventually become far less common than they are today. Had the opposite decision – that `using` allowed unqualified functions to define members of a namespace – been taken, a programmer would never have been sure whether an apparently global function really was global. Its definition could have been captured by some unknown declaration in some namespace.

Typically, only a subset of the declarations in an implementation are part of the header(s) given to users as the interface(s). The implementor will therefore put additional names into a namespace and will also gain access to additional information from other name spaces needed by the implementation (only). For example:

```
#include "my_library.h" // interface part of library namespace

namespace my_library { // implementation details
        // ...
}

using namespace my_library; // convenient access to all of my_library
```

```
#include "helper1"
#include "helper2"

using namespace helper1;        // or apply ‘‘using’’ selectively
using namespace helper2;        // or apply ‘‘using’’ selectively

// ...
```

This kind of distributed specification of members of a namespace can be used to supply several header files for a system (describing different aspects of it) without having to use several different name spaces.

Alternatively, implementors may choose to keep implementation details in their own namespace:

```
#include "my_library.h"

namespace my_library_impl { // implementation details

        using namespace my_library;
        // ...
}

using namespace my_library_impl; // convenient access to all of my_library

#include "helper1"
#include "helper2"

using namespace helper1;        // or apply ‘‘using’’ selectively
using namespace helper2;        // or apply ‘‘using’’ selectively

// ...
```

This makes the separation between the interface and the implementation clearer and will therefore often be the better implementation technique.

4 Multiple Namespaces

Consider two namespaces using the same name:

```
namespace A {
        class String { /* ... */ };
        // ...
}

namespace B {
        class String { /* ... */ };
        // ...
}

using namespace A;
using namespace B;

String s; // error: A::String or B::String
```

Clearly, the unqualified use of `String` is an error. However, should it be an error to say

```
using namespace A;
using namespace B;
```

when `A` and `B` both have a `String`? That is, do we check for inconsistent *using-directives* or inconsistent uses? We check for inconsistent uses (only). This is in line with other places in the language – for example, overloading and name resolution where multiple inheritance is used – where we outlaw actual errors only, and not potential errors. Also, a user that worries about potential name clashes can minimize such clashes by being more selective:

```
using A::String;
using B::f;

String s; // ok: A::String
```

Note that using *using-declarations* lead to clashes at the point of the *using-declarations*:

```
using A::String;
using B::String;        // error: Two definitions of String
class String { ... };   // error: Three definitions of String
```

One way of thinking of this is that a *using-directive*

```
using namespace A;
```

does not enter anything into the local scope. A more specific *using-declaration*

```
using A::(f, g);
```

actually enters `f` and `g` into the local scope. Naturally, implementors have a variety of techniques at their service so this may not actually be the way an implementation really works – it will simply look that way to a user.

So, how does a user resolve a clash caused by multiple *using-directive*s?:

```
using namespace A;
using namespace B;

String s; // error: A::String or B::String
```

One way would be to modify the code:

```
A::String s;
```

but that is not always feasible or convenient. The alternative of changing the *using-directive*s to more specific *using-declaration*s was mentioned above, but there is a third alternative: Use a *using-declaration* in an extra scope to resolve the ambiguity:

```
using namespace A;
using namespace B;

namespace Mine { // namespace introduced to allow declaration of String

        using A::String;

        String s; // A::String

        // ...
}
```

Multiple *using-directives* shouldn't be overused. They are essential for straight forward conversions of older code into namespaces and for compatible use of standard libraries. However, in most cases easier to maintain code can be obtained by minimizing and localizing the use of *using-directives*.

Old Code

I expect that it will be common to take code that does not use namespaces and convert it to use namespaces. An old program or an old library will typically consist of a couple of header files containing class declarations, constants, templates, inline functions, etc., and a set of `.c` files containing the definition of functions, global variables, etc.

The conversion can be done only by changing the source code and (re)compiling. The first step in the conversion would be to wrap the header files in namespaces. In doing so, we must exclude `include`s of other headers and declarations referring to functions, types, variables, etc. defined elsewhere. For example:

```
// Mine.h:
// not using namespace

#include <iostreams>
#include "foob.h"

extern int a;
const int c = 7;
// ...
extern g(); // not one of mine; I'm just using g()
// ...
void f();
inline int frob() { return c+99; }
```

becomes

```
// Mine.h:
// now namespace Mine

#include <iostreams>
#include "foob.h"

namespace Mine {
        extern int a;
        const int c = 7;
        // ...
} // namespace Mine

extern g(); // not one of mine; I'm just using g()

namespace Mine {
        // ...
        void f();
        inline int frob() { return c+99; }
} // namespace Mine
```

I like the indentation, but realistically, it will often not be done when converting hundreds of lines of declarations. Had distributed specification of members of a namespace not been allowed we would have had to reorder this header.

Next we need to ensure that the definitions in the `.c` files match up with the declarations now in the namespace `Mine`. This can be done by any of the techniques mentioned in §4. However, someone doing a conversion that is not part of a significant rewrite will want to minimize the modifications to the code. This can be done by wrapping the files, excluding declarations of supporting functions, variables, etc., from elsewhere, in

```
namespace Mine {
        // ...
}
```

This ensures that functions that are not named in the header `Mine.h` don't escape into the global name space.

I expect that it will be common to have two versions during a transition period: One for systems that supports namespaces and one for older systems that don't. This can be managed either by totally separate sources or by using `#ifdef` to separate out the `namespace` and `using` constructs.

5 Name Resolution

The naming rules associated with this name space proposal are designed to provide a user a choice between notational convenience and safety in the access to functions in a name space. Consider:

```
        // my old program:

        void f(double);

        void g()
        {
                f(1);    // calls f(double)
                f(1.0);  // calls f(double)
        }
```

Let us introduce a namespace A into this program:

```
        // my old program:      // now modified

        void f(double);

        namespace A {
                void f(int);
        }

        using namespace A;  // make A's names available

        void g()
        {
                f(1);    // calls f(double)     // no, now calls f(int)!
                f(1.0);  // calls f(double)     // still
                A::f(1); // calls f(int)
        }
```

The namespace was not only defined but also made accessible (by the *using-directive*). This implied a change of meaning of the program (according to the rules of overload resolution) exactly as would have been the case if the function `f(int)` had been declared without the name space mechanism. This is correct and desirable behavior if you consider the functions from A at the same logical level as the global functions. This will typically be the case when all of the global functions are in fact imported from some name space or other or if the global functions are intended to supplement the functions supplied by a library.

Prefer Global Names

However, the names from a namespace will often be the interface to a library and simply included by some `#include` directive by a user that never actually looked carefully at the contents of the included file. In that case, the user might want to give priority to global functions and then it makes sense to use functions from a name space only in their qualified form:

```
        // my old program:  // now modified

        void f(double);

        namespace A {
                void f(int);
        }

        // no ''using namespace A;''

        void g()
        {
                f(1);     // calls f(double)  // still
                f(1.0);   // calls f(double)  // still
                A::f(1);  // calls f(int);
        }
```

This can be useful where a minor modification is made to an existing program and preservation of behavior is essential. However, being explicit for some calls only can lead to surprises because the overload resolution mechanism has been effectively disabled.

Prefer ‘‘Own’’ Names

Another approach involves wrapping the ‘‘local’’ code into a namespace to ensure that ‘‘local’’ declarations are given priority over names from other namespaces:

```
// my old program:      // now modified

namespace A {
        void f(int);
}

namespace Mine {

        void f(double);
        using namespace  A;

        void g()
        {
                f(1);    // calls f(double)
                f(1.0);  // calls f(double)
                A::f(1); // calls f(int)
        }
}
```

This technique of wrapping one’s own functions in a namespace has three important properties:

[1] You no longer pollute the global space with your own names.

[2] A change in included namespaces such as `A` no longer affects working code (even where *using-directive*s are used) because the local declarations are given priority.

[3] It allows me to be explicit about what names are my own. Note that global names comming from include files are often not controlled by their users so considering all global names ‘‘my own’’ would often be a poor assumption.

Again, the technique of wrapping one’s own code in a namespace uses the scope rules to limit the effectiveness of the ambiguity control mechanism and can therefore be dangerous. An alternative view is that disabling overloading in this way enhances safety.

Overload Resolution

Note than a *using-declaration* introduces every function of a given name into a new scope. This is essential to preserve the designer’s intentions about their use. For example:

```
namespace A {
        void f(int);
        void f(char);
}
```

It is clearly the intent of the designer of `A` that `f(97)` should call `f(int)` and `f(’a’)` should call `f(char)`. Allowing selective inclusion of `f(int)` and `f(char)` into a scope would lay the user open to subtle and unnecessary errors. For example:

```
void g()
{
        using A::f(int);  // not allowed, but ‘‘what if?’’
        f(’a’);           // calls f(int), probably wrong
}
```

When converting old code it is essential that overload resolution apply across namespace boundaries. Consider:

```
// my.h:
        int f(int);
        // ...
// your.h:
        double f(double);
        // ...

// x.c
        #include my.h
        #include your.h

        // ...

        void g()
        {
                f(1);    // calls f(int)
                f(1.0);  // calls f(double)
        }
```

After the obvious conversion to namespaces this will work as before:

```
// my.h:
        namespace my {
                int f(int);
                // ...
        }

// your.h:
        namespace your {
                double f(double);
                // ...
        }

// x.c
        #include my.h
        #include your.h

        using namespace my;
        using namespace your;

        // ...

        void g()
        {
                f(1);    // calls f(int)
                f(1.0);  // calls f(double)
        }
```

Similarly, using a namespace for `my.h` only will not change the meaning.

C Struct Hack

I do not propose to extend the C compatibility hack of allowing the same name for a class and a non-class names to apply across name spaces. For example

```
class X { public: /* ... */ X(); /* ...*/ };
void X(int);

void f()
{
        struct X a;
        X(1);         // function call
}
```

This cannot be rewritten as

```
namespace A {
        class X { public: /* ... */ X(); /* ...*/ };
}
namespace B {
        void X(int);
}

using namespace A;
using namespace B;

void f()
{
        struct X a;   // error
        X(1);         // error
}
```

The reason to disallow this is partly a wish not to perpetuate the C struct hack, partly to ensure that a C++ syntax checker isn't required to do ambiguity resolution. Consequently, a *use* of a name that is found as a type name in one namespace and as non-type name in another is an error.

Dispersed namespace definitions

Consider:

```
namespace A { void f(); }

using namespace A;

void f1()
{
        g();    // error: no g() declared
}

namespace A { void g(); }

void f2()
{
        g();    // ok? ok!
}
```

The second call of `g` is ok because ''`using namespace A;`'' is a directive to look into `A` whenever a name lookup is done. It is *not* a declaration that enters names from `A` into the current scope. On the other hand

```
namespace A { void f(); }

using A::g;
```

is an error because there is no `A::g` to refer to. Similarly, definition of names not already declared in a namespace is not allowed. For example:

```
void A::g() { /* ... */ } // error: no g in A
```

The reason not to allow addition of a new name into a namespace `A` simply by defining it using `A::` is to protect against misspellings, misunderstandings as to which namespace a name belongs to, etc.

Lookup Details

Using a namespace name before it has been declared is not allowed in order to make it possible to catch typos. For example:

```
// no previous definition of A
using namespace A;  // error: A undefined
```

If necessary, the effect of a forward declaration can be achieved with dispersed namespace definition:

```
namespace A { }
using namespace A;      // fine
// ...
namespace A { /* ... */ }
```

This makes it possible to write mutual references between namespaces:

```
namespace B { }

namespace A {
        using namespace B;
        // ...
}

namespace B {
        using namespace A;
        // ...
}
```

Fortunately, it is trivial for implementations to avoid infinite loops by simply never `using` the same namespace twice in a single lookup.

Global Scope

Consider

```
int a;

void f()
{
        int a;
        a++;     // local a
        ::a++;   // global a
}
```

If we wrap a namespace around this and add yet another variable called `a` we get:

```
int a;

namespace X {
        int a;

        void f()
        {
                int a;
                a++;    // local a
                X::a++; // X::a
                ::a++;  // X::a or global a ? global a !
        }
}
```

In other words, we have to decide whether qualification by `::` means ‘‘global’’ or ‘‘in the nearest enclosing namespace.’’ The latter would ensure that wrapping arbitrary code in a namespace implied no change of meaning. However, that would leave the genuinely global name inaccessible. Consequently, I propose the former meaning. Therefore, `::a` refers to a name found in the global scope (including, of course, names imported into the global scope by *using-declaration*s and *using-directive*s).

Note that *using-directive*s can cause clashes between global names and names in a namespace:

```
int a;

namespace X {
        int a;
}

int i1 = a; // global a

using namespace X;

int i2 = a;   // error: X::a or global a ?
int i3 = ::a; // error: X::a or global a ?
```

6 Nested Namespaces

One obvious use of namespaces is to wrap a complete set of declarations and definitions in a separate name space:

```
namespace X {
        // all my declarations
}
```

The list of declarations will in general contain namespace declarations. Thus, for practical reasons – as well as for the simple reason that constructs ought to nest unless there is a strong reason for them not to – nested namespaces are allowed. For example:

```
namespace X {
        namespace Y {
                void f();
                // ...
        }
        // ...
}
```

A user of `X` can use names from `Y` only through explicit qualification or an appropriate `using` declaration:

```
f();                  // error: no f in global scope
X::f();               // error: no f in X
X::Y::f();            // fine

using namespace X;
Y::f();               // fine

using namespace Y;
f();                  // fine
```

or alternatively

```
using namespace X::Y;
f();                  // fine
```

Naturally, `using` can be used within a namespace. For example:

```
namespace A {
        void f();
}
```

```
namespace B {
        namespace C {
                void g();
        }
        void h();
        using namespace A;  // make A’s names visible through B
        using namespace C;  // make C’s names visible through B
}

void k()
{
        using namespace B;
        f();    // B::f == A::f
        g();    // B::g == B::C::f
        h();    // B::h
}
```

This implies that namespace names are type names from the point of view of a parser and thus not a new implementation burden.

Note that `using` a namespace `A` from within a namespace `B` does not make `A` itself a member of `B`:

```
namespace A {
        void f();
}

namespace B {
        using namespace A;  // make A’s names visible through B
        void f();
        void g();
}

void h()
{
        B::A::f();         // error: No A in B
        B::f();            // ok
        using namespace B;
        f();               // error: ambiguous
}
```

The reason that `B::f` doesn’t lead to an ambiguity with `A::f` is that `f` is found in `B` so that there is no need to look out into the global scope – where `A::f` would have been found because of the `using namespace A`.

To contrast, the plain `f()` is ambiguous because the `using namespace B` in `h()` plus the `using namespace A` in `B` implies that both `A::f` and `B::f` are seen when the global scope is considered looking for `f()` outside `h()`.

Using a non-local name from within a namespace is equivalent to having qualified it with the namespace name when looking:

```
void B::g()
{
        f();  // ok: B::f
}
```

As seen from `B::g()`, `f` is the local name `B::f` so there is no need to look ‘‘out’’ into the global scope – where `A::f` would have been found because of the `using namespace A`.

7 Namespace Names

The name of a global namespace must be unique among the global names in a program. For example:

```
namespace Foo {
        // ...
}

void Foo();     // error: name clash
```

This implies that the problem of name clashes has been ‘‘moved out’’ one level rather than solved in general. This is acceptable because it reduces the number of names that can clash by a couple of orders of magnitude and also because there is a class of non-clashing names that can be used for namespace names. I imagine that company names and names of major libraries will be popular as namespace names. Such names are kept distinct by commercial law and also by strong interests in keeping product names separate and widely known. A namespace called `bs` might work in a local environment but would be ill-chosen for world-wide use. However, `ATT`, `Rational`, and `IBM` will only clash provided there already are other problems with those names†. Allowing the user to disambiguate clashing names in the absence of source code requires tools that cannot be portable because they will have to operate on object code format and obey local linkage conventions. However, in many environments it is not too hard to write a tool that converts an object file to another that is equivalent except that different names are used. This can allow a particular user to overcome problems that cannot be solved in general.

Using longer names will help avoid clashes, but more or less requires the use of synonyms:

```
namespace American_Telephone_and_Telegraph {
        // ...
}
namespace ATT = American_Telephone_and_Telegraph;
```

Longer names carries a – presumably not too large – overhead given the current state of common linker technology.

One might consider avoiding clashes between namespace names and other names by requiring uses of namespace names to be prefixed by the keyword `namespace` much as C structure tags have to be prefixed by the keyword `struct`. I consider that too ugly. To see that, rewrite the examples in this paper to consistently use `namespace` as a prefix or read a large C program that does not use typedef or `#define` to minimize `struct` as a prefix.

Note that the name of a namespace can be ‘‘re-used’’ within a namespace, but such a name will cause a name clash if used unadorned in the global scope. For example:

```
namespace X {
        class X {
                // ...
        };
        // ...
}

X::X a;   // no problem so far

using namespace X;  // no problem so far

X b;      // error: namespace X and class X used in same scope
```

Naturally, short simple minded names such as `Mine`, `X`, and `A` should be avoided in real code.

† There are – or at least there used to be – two companies with the legal name ‘Rational’ in the USA.

8 Standard Libraries

Consider:

```
#include <iostream.h>
#include <string.h>

int main()
{
        char* p = "Hello, world!"
        cout << p << ' ' << strlen(p) << '\n';
}
```

This had better work as expected. However, we would also like to use iostreams and C-style strings without polluting the global namespace.

This can be done like this: First take the current `<string.h>` and use it to write a new header that we might call `_string.h`:

```
// _string.h:

        namespace Clib {
                extern "C"
                        // contents of C <string.h>
                }
        }
```

and use this to define a new `string.h`:

```
// string.h:

        #include <_string.h>
        using Clib;
```

This assumes that all standard C libraries are in a single namespace and have C linkage, but that is not essential for this discussions.

Given this, the program above will compile – meaning that we haven't broken any old code. Someone who wants to use C-style strings without polluting the global name space might write:

```
#include <iostream.h>
#include <_string.h>

int main()
{
        char* p = "Hello, world!"
        cout << p << ' ' << Clib::strlen(p) << '\n';
}
```

Note that nested includes are very common. In general, this implies that either the headers must contain *using-directive*s or else the users must provide *using-directive*s for indirectly included namespaces. For example:

```
// h1.h:
        #include "h11.h"
        #include "h12.h"
        namespace h1 {
                // ...
        }
```

Here the user might have to say

```
include "h1.h"
using namespace h1;
using namespace h11;
using namespace h12;
```

This will in some cases be a violation of an abstraction because the user sees `h1` as a single set of services and not as a composite. If so, `h1.h` can be rewritten to

```
// h1.h:
        #include "h11.h"
        #include "h12.h"
        namespace h1 {
                using namespace h11;
                using namespace h12;
                // ...
        }
```

to allow

```
include "h1.h"
using namespace h1;
```

Namespaces and C Linkage

Consider:

```
namespace A {
        extern "C" void f(int);
        extern "C" void g(int);
}

namespace B {
        extern "C" void f(int);
        extern "C" void g(double);
}
```

Is this legal? Does `A::f()` and `B::f()` refer to the same C function? After all, there cannot be two C functions called `f()`. To answer these questions let's first consider a couple of simple cases:

```
namespace A {
        void f(int);
        void g(int);
}

namespace B {
        void f(int);
        void g(double);
}
```

This is clearly legal and `A::f()` and `B::f()` are different functions. How about?:

```
namespace A {
        extern "Ada" void f(int);
        extern "Ada" void g(int);
}

namespace B {
        extern "Ada" void f(int);
        extern "Ada" void g(double);
}
```

Since Ada allows function name overloading and has a package concept that resembles namespaces one would expect this is to legal and `A::f()` and `B::f()` to refer to different Ada functions. Anything else would require assumptions about the semantics about Ada – and a C++ compiler cannot be assumed to know Ada. Thus, as far as C++ is concerned `A::f()` and `B::f()` refer to different functions independently of linkage specifications.

Then what about C? The analogy to ordinary C++ and linkage to ‘‘other languages’’ indicates that `A::f()` and `B::f()` must be considered different functions by C++ even if they have C linkage.

At first glance, this appears to prevent the use of namespaces for things like the C++ versions of the C standard libraries – an unacceptable restriction. Consider:

```
namespace Clib {
        extern "C" int strlen(const char*);
}
```

To be useful, `strlen` must link to the `strlen` found in the standard C library. This can be achieved by either having `Clib` be ‘‘magic’’ or by having C linkage imply that the linkage name is the same as a global name even for functions and objects declared within namespaces. Making `Clib` ‘‘magic’’ doesn’t sound like a very good idea. For starters, we would need several such ‘‘magic’’ namespaces.

Fortunately, the other alternative appears manageable. C++ would consider functions such as `A::f()` and `B::f()` to be distinct. However, if more than one had C linkage a linker error would occur. In general, a C++ compiler wouldn’t be able to detect such errors except where `A::f()` and `B::f()` was actually defined with C linkage in the same file.

9 Eliminating Global `static`

It is often useful to wrap a set of declarations in a namespace simply to avoid interference from declarations in header files or to avoid having the names used interfere with global declarations in other compilation units. For example:

```
#include <header.h>
namespace Mine {
        int a;
        void f() { /* ... */ }
        int g() { /* ... */ }
}
```

However, in such cases we aren’t really interested in the name of the namespace as long as it doesn’t clash with other namespace names. To serve that need more elegantly we could allow a namespace to be unnamed:

```
#include <header.h>
namespace {
        int a;
        void f() { /* ... */ }
        int g() { /* ... */ }
}
```

Except for overloading by names in the header, this is equivalent to

```
#include <header.h>

static int a;
static void f() { /* ... */ }
static int g() { /* ... */ }
```

Such overloading is usually undesirable and also easily achievable when desired:

```
namespace {
#include <header.h>
        int a;
        void f() { /* ... */ }
        int g() { /* ... */ }
}
```

Therefore, I propose that we deprecate the use of `static` for control of visibility of global names. That would leave `static` with a single meaning: statically allocated, don’t replicate.

An unnamed namespace is unique to its compilation unit. A name from an unnamed namespace can be accessed without qualification after its point of declaration.

10 Implications for Classes

It has been suggested that a namespace should be a kind of class, but I don't think that is a good idea (see Appendix B). The opposite, that a class is a kind of namespace, seems almost obviously true. A class is a namespace in the sense that all operations supported for namespaces can be applied with the same meaning to a class unless the operation is explicitly prohibited for classes. This implies simplicity, generality, while minimizing implementation effort.

Derived Classes

Let us consider some implications:

```
class B {
public:
        f(char);
};

class D : public B {
public:
        f(int);
};

void f(D& d)
{
        d.f('c'); // calls D::f(int)
}
```

This resolves as ever. There is a new explanation, though: `D` is a namespace. The namespace `D` is nested in the namespace `B` so `D::f(int)` hides `B::f(char)` so `D::f(int)` is called.

Now if you don't like that you can try this:

```
class B {
public:
        f(char);
};

class D : public B {
public:
        f(int);
        using B::f;     // bring B::f into D and enable overloading
};

void f(D& d)
{
        d.f('c'); // calls D::f(char) !
}
```

We suddenly have a choice.

As ever, names from different sibling base classes clash (independently of what they name):

```
struct A { void f(int); };
struct B { void f(double); };

struct C : A, B {
        void g() {
                f(1);    // error: A::f(int) or B::f(double)
                f(1.0);  // error: A::f(int) or B::f(double)
        }
};
```

`Using` Base Classes

To avoid confusion a *using-declaration* that is a class member must name a member of a (direct or indirect) base class. To avoid problems with the dominance rule *using-directives* are not allowed as class members.

```
        struct D : A {
                using namespace A; // error: using directive as member
        };
```

Also consider:

```
        class B {
        public:
                f(char);
        };

        class D : private B {
        public:
                using B::f;
        };
```

This achieves what currently requires use of an access specification:

```
        class D : private B {
        public:
                B::f;
        };
```

Thus, `using` declarations make access specifications redundant. Consequently, I propose to deprecate them.

`Using` Classes

Consider:

```
        class X {
                static void f();
                void g();
        };

        void h()
        {
                using namespace X;  // ok?
                f();
                g();  // error: object missing for member function
        }
```

That is, should it be allowed to specify a *using-declaration* or a *using-directive* for a class? My suggested answer is ‘‘yes.’’ My (weak) argument for ‘‘yes’’ is ‘‘why not?’’

One possible use is to replace `friend` functions with `static` members in some cases:

```
        class complex {
                // ...
                static complex operator+(complex,complex);
        };
        complex a, b;
        complex z1 = a+b;  // error no complex + in scope

        using namespace complex;
        complex z2 = a+b;  // ok
```

I remain less than completely convinced about the utility of this technique.

11 Definition of Namespaces Features

This is a first cut at a reference manual style definition of namespaces.

To integrate namespaces into the reference manual text, namespaces must be defined before classes and the aspects of classes that relates to name lookup must be rephrased in terms of namespaces.

Namespaces

The grammar for the namespace constructs is:

namespace-definition:
 `namespace` *identifier*$_{opt}$ `{` *declaration-list*$_{opt}$ `}`

namespace-alias-definition:
 `namespace` *identifier* `=` *namespace-name* `;`

using-declaration:
 `using` *namespace-name* `::` *identifier* `;`
 `using` *namespace-name* `::` `(` *using-list* `)` `;`

using-directive:
 `using namespace` *namespace-name* `;`

namespace-name:
 identifier
 namespace-name `::` *identifier*

using-list:
 name
 using-list `,` *name*

An *identifier* in a *namespace-name* must have been specified to refer to a namespace by being the *identifier* in a *namespace-definition* or a *namespace-alias-definition*.

The *identifier* in a *namespace-alias-definition* may not be defined elsewhere in its scope and defines a synonym for the *namespace-name* mentioned. The *identifier* in a *namespace-definition* must either be undefined in its scope or be a *namespace-name* defined in a *namespace-definition* in the same scope. All namespaces with the same *namespace-name* in the same scope are considered part of the same namespace. All global namespaces with the same *namespace-name* are considered part of the same namespace. An *identifier* used as a global *namespace-name* cannot be used as the name of any other global namespace, template, type, function, object, or value in the program.

A *namespace-definition* or a *namespace-alias-definition* is a *declaration*. The *identifier*s mentioned after `::` in a *using-declaration* must have been declared in the namespace named by the *namespace-name.*

A *namespace-definition* can only be occur at the global scope or within another namespace. A *using-declaration* can be used as a *declaration-statement*, a *member-declaration*, at the global scope, or within another namespace. A *using-directive* can be used as a *declaration-statement*, at the global scope, or within another namespace (that is not a class).

All unnamed namespaces within a compilation unit are treated as one namespace ‘‘the unnamed namespace’’ and treated as if it had a name that is unique in a program. A name from an unnamed namespace can be accessed without qualification after its point of declaration.

Members of a namespace can be defined within that namespace. Members of a named namespace can also be defined using explicit qualification.

An `extern` or `friend` function first declared within a namespace or a non-namespace member of a namespace is a member of that namespace. A type name first declared within a namespace or a non-namespace member of that namespace is a member of that namespace.

Comments on the Grammar

Note that a *name* need not be a simple *identifier*; for example, it can be an *operator-name*.

Silly typing errors will inevitably arise from the syntactic similarity of the namespace constructs to other C++ constructs. I propose we allow an optional semicolon after a global declaration to lessen the frustration. This would be a kind of ‘‘empty declaration’’ to match the empty statements.

The keywords `namespace` and `using` were chosen in the hope of minimizing clashes with identifiers. The word `extern` could be used instead of `namespace` to save a keyword, but this kind of keyword overloading has led to confusion in the past.

I considered several alternatives for a syntax for specific *using-declarations*. The minimal syntax is simply a *namespace-name* followed by a list of *identifiers*:

```
using A f String g
```

However, that seemed too minimal, too error-prone, and out of line with the way lists are represented elsewhere in the C++ grammar so after a few experiments I settled the namespace name followed by `::` followed by either a single name or a comma separated list enclosed in parentheses:

```
using A::f;
using A::(f, String, g);
```

The former is obvious; the latter simply avoids repetition of the `A::`. The types of the names `f`, `String`, and `g` are found in `A`.

The simpler alternative of allowing a single name only was considered attractive, but restrictive.

The mention of `namespace` in a *using-directive* is redundant. A compiler could distinguish *using-directive*s from *using-declarations*s based on the type of the name specified. However, that would complicate parsing – and more importantly – I found the more explicit form easier to teach and read.

Explicit Qualification

A name defined in a namespace can be accessed qualified by a *namespace-name* for its namespace using the `::` operator . A qualified name can be used as a *dname* in a definition. In that case, the definition from the point of the qualification until the end of the declaration is considered in the scope of the namespace. For example:

```
namespace X {
        typedef int I;
        I f(I);
}

X::I X::f(I a) { /* ... */ }   // correct
I X::f(I a) { /* ... */ }      // error: return type not in scope
```

`using`

The member names specified in a *using-declaration* are defined in the scope in which the *using-declaration* appears. The names thus declared are aliases for their original declarations so that the *using-declaration* does not affect the type, linkage, etc. of the members referred to. A *using-declaration* is not a definition, thus redundant *using-declaration*s of a name are allowed.

A *using-directive* specifies that the names in the namespace can be used in the scope in which the *using-directive* appears exactly as if the names from the namespace had been declared outside a namespace at the point where their namespace was declared.

A *using-declaration* or a *using-directive* does not affect names being declared.

Classes

The scope of a class is a namespace. The scope of a derived class is nested in the scope of each of its base classes. Qualification and *using* apply to classes exactly as to (other) namespaces. A *using-declaration* used as a member declaration must refer to accessible base classes and/or accessible members of base classes. A namespace that is not a class cannot be defined within a class. A namespace that is a class cannot have names added by further namespace declarations.

The *access-specifier* syntax is deprecated.

Comments on Classes
Note that a *using-directive* cannot be used as a member declaration.

Names cannot be injected into a class by friend declarations, etc., but they can be injected into a namespace; see appendix E.

A namespace is open; that is, you can add names to it from several namespace declarations. Classes are not.

`Static`
The use of `static` at the global scope or within a namespace that is not a class is deprecated.

12 Implementation and Compatibility Issues

An implementation of this proposal can be done in the compiler only. Basically, the link-time aspects namespaces can be implemented as a formalization of the old add-a-prefix-to-names techniques for avoiding global name clashes. No linker support beyond support for long names is necessary. On the other hand, linker support can provide more convenient and potentially more efficient alternatives to the use of long names.

It is possible to implement namespaces without breaking link-compatibility.

No run-time support is needed.

The implementation of `using` implies that the name lookup mechanism will have to search multiple scopes (namespaces) when looking for declarations and apply the overloading rules to names from different name spaces. This strongly resembles what has to be done for lookup in multiple inheritance hierarchies so C++ compilers already have the internal mechanisms needed to support `using`.

Except for clashes with the new keywords `namespace` and `using` there should be no source compatibility problems. Placing standard libraries in namespaces could have compatibility implications, but it seems that placing suitable `using` declarations in standard headers will ensure that no existing program changes its meaning.

Experience shows that this proposal can be implemented in five days. The amount of code added was of the order of 300 lines of C++. Our measurement tools wasn't precise enough to detect a difference in compile time speed caused by namespaces. We haven't yet measured link-time overheads.

13 Acknowledgements

This proposal has its origins in discussions on name space control that has taken place over the last couple of years at standards meetings, conferences, and in particular on the extension group mail reflector. Here is a list of people I know contributed to the discussions or commented on previous versions of this paper: Dag Brück, John Bruns, Reg Charney, Steve Dovich, Bill Gibbons Philippe Gautron, Tony Hansen, Peter Juhl, Andrew Koenig, Eric Krohn, Doug McIlroy, Richard Minner, Martin O'Riorden, John Skaller, Jerry Schwarz, Mark Terribile, and Mike Vilot. Peter Juhl wrote the initial implementation.

14 Appendix A : A Simpler Alternative

I have tried to be comprehensive, and that always leads to a perception of complexity. After all, you don't have to have seen the really warped examples to use an language feature. However, the `namespace` and `using` proposal *is* more complex than I would have liked and several further extensions has been suggested (and rejected).

Three simplifications have also been suggested:

[1] Making a namespace a kind of class.

[2] Don't have a `using` declaration.

[3] Making a namespace declaration imply `using`.

Option [1] is discussed in Appendix B below. This section considers [2] and [3].

No using

Consider [2]; that is, a namespace is defined as described above, but to use a namespace we must use explicit qualification (there is no `using` declaration).

This makes it significantly harder to use a namespace than not to use it. Consider having to write `Clib::` in front of every use of a standard C library function. I think that this would be intolerable and would lead to demands that ‘‘common and important’’ libraries should be ‘‘special’’ in the sense that their names should be ‘‘truly global.’’ Unfortunately, everyone would demand that privilege for their library and sloppy practice would prevail. It would be too much of a convenience to ignore the `namespace` feature. I have no faith in our ability to preach ‘‘good manners’’ where ‘‘bad manners’’ provide a significant convenience.

Some convenience could be achieved through the introduction of aliases:

```
namespace X {
        // ...
}

typedef X::String string;
int& v = X::v;
```

However, if such declarations find their way into common use (especially if they appear in standard header files) we have lost the benefits of namespaces by explicitly polluting the global name space again. If instead we made the introduction of local synonyms easier we would have reinvented `using`.

Implicit using

Consider [3]; that is, a namespace defined is as described above, but the declaration of a namespace implies a `using` so that every name is accessible directly by its non-qualified name.

Not having an explicit `using` declaration is a significant simplification, but we lose the ability to selectively include names from a namespace.

```
namespace X {
        class String { /* ... */ };
        int f(int);
}

namespace Y {
        class String { /* ... */ };
        double f(double);
}

void f()
{
        String s = "asdf";      // error: ambiguous
        X::String s = "asdf";   // ok
        Y::String s = "asdf";   // ok
        typedef X::String String;  // from now on String means
                                   // X::String

        f(1);    // call X::f(int)
        f(1.0);  // call Y::f(double)
        Y::f(1);    // call Y::f(double)
        X::f(1.0);  // call X::f(int)
        int (&f)(int) = X::f;  // from now on f means
                               // X::f
}
```

Clearly, the ambiguity control provides some protection against surprises and `typedef`s and references can be used to compensate for the lack of selective `using` declarations. However, it cannot be guaranteed that adding a namespace to an existing program doesn't change the meaning. Consider:

```
int f(int);

// ...

void h()
{
        f(1.0);  // calls f(int)
}
```

Adding a namespace Y might change its meaning (as specified by the overload resolution rules):

```
int f(int);

// ...

namespace Y {
        double f(double);
        // ...
}

// ...


void h()
{
        f(1.0);  // calls f(int)  // no longer:
                                  // now it calls Y::f(double)
}
```

Is this an acceptable cost to make saving us from `using` declarations? Many have expressed the opinion that it isn't: It must be guaranteed that names doesn't ‘‘leak’’ from a namespace into the global namespace without explicit programmer action.

Note that because the lookup rules for `namespace`s with implicit `using` are almost identical to the lookup rules for `namespace`s plus explicit `using` declarations, there isn't a significant difference in the implementation difficulty of the two variants.

15 Appendix B: Namespaces and Classes

A class is a type. A class is also a mechanism for grouping declarations in a separate scope. As shown in §1 a class can be used for name management though it is not ideal. However, could the class concept be extended to become a convenient mechanism for name space control thus rendering `namespace` redundant or allowing `namespace` to be defined as a kind of class? Actually, of course it *could*, the real questions are ‘‘should it?’’ and ‘‘how different from other classes would a `namespace` have to be?’’

I don't have a ‘‘killer argument’’ for or against namespaces as classes, but my view is that on balance equating namespace and class is more trouble than it is worth; that is why this section is an appendix and not a proposal.

Having a `namespace` be a kind of `class` in a manner similar to the way `union`s and `struct`s are `class`es might minimize the set of concepts a user had to learn and maximize the uniformity of the language rules. So consider defining a `namespace` as a `class` from which no objects can be created, but that is an object itself (much like an anonymous `union` is). To ease discussion, let us call such a class a `module` to distinguish it from `namespace` as discussed above. However, if we actually decided to go with a namespace as a class variant I think that ‘‘namespace’’ might still be the better keyword.

Thus defined, a `module` need not have the restrictions on membership specified for other classes to make object creation manageable. For example, a `module` can have template members (a necessity). A `module` can't have operator `new` and `delete` functions because there is only one (implicitly allocated) object of the `module` ‘‘type.’’ However, a `module` constructor might be useful, as might a `module` destructor. Please note, however, that a `module` could only have one constructor and that would almost certainly have to be a default constructor. A `module` might also have virtual functions, though, because there seem no reason not to give the unique object representing the `module` a virtual function table.

Derived `module`s could provide overriding functions.

Name Lookup
We could give a module ‘‘implicit `using`’’ as suggested in variant [3] above, but that is out of character for a class, so we must consider if a `using` declaration is needed. It is needed for the reasons given in the discussion of variant [2] above. In other words, lookup issues are not affected by having a `module` as class compared to the `namespace` (non-class) proposal.

Distributed Specification
The key difference is that a class (interface) is specified in one place and a `namespace` as specified above can be added to wherever necessary. Thus a `module` has a unique (centralized) interface whereas a `namespace` does not specify a limited set of names defined in a unique place.

Is this different important and if so in which ways? At first glance, *not* being distributed looks like a serious problem. However, derived modules and/or `using` declarations can be used to provide most benefits of extensibility. For example, say module `X` is given and I really want to add to it:

```
module X {
        // ...
};
```

I can simply derive from `X`:

```
module Y : public X {
        // ...
};
```

Anything within `Y` can now use `X`’s public and protected names as conveniently as `X`’s own functions, and my users can use `X`’s and the names I added through `Y` as conveniently as if I had added to `X` directly.

The same effect can be achieved through `using`:

```
module Y {
        using namespace X;
        // ...
};
```

In fact, this is the way one would simulate inheritance in the `namespace` proposal.

We generally try to avoid having object definitions in headers, especially objects that are meant to be unique. Since a `module` defines a unique object, yet it will typically live in a header to allow its use, `module` definitions would be unique in the language (and thus different from other classes and a likely source of controversy and confusion). The implementation support for `module`s would be unique also. However, any technique used to obtain a unique virtual function table can be used.

Are there any definite and significant benefits from having a unique definition of a module compared to the distributed definitions of a namespace? I see none. If they exist, they ought to have something to do with initialization (see below).

Class specific Features
Let’s consider class specific features to see if they help in name space management. Constructors and destructors help with initialization and cleanup, but there is no really good way to specify initialization order for global objects in different compilation units. Maybe the module concept could help? Consider:

```
// X.h
        module X {
                A a;
                B b;
        };

// X.c
        B B::b = g(/*args*/);
        A X::a = f(/*args*/);
```

Ordinary class rules say that member `a` should be constructed before member `b`. However, ordinary global object initialization rules say that `B::b`'s initializer should be executed before `X::a`'s. We could resolve this (one way or the other or as an error), but having a namespace as a class actually added a problem rather than helping us by providing a existing rule to resolve the issue. Further, unless we introduce something new (not already in our class concept) the module concept does not help with initialization order problems.

Derived modules and virtual module functions seems an intriguing concept. However, derived modules can be simulated trivially by namespaces and `using`, and I have trouble seeing any important uses of `module` virtual functions. All I can come up with for `Module`s seems to be just as well handled by a class with a unique object:

```
class X { /* ... */ } X;
```

The essential power of derived classes comes when objects are manipulated through interfaces defined as base classes referred to through pointers and references. Having a single anonymous object only cramps my style.

Classes interact with templates and I can think of some really nice uses for template `module`s and for `module` names as template arguments. However, there is nothing particularly difficult in extending the template concept to allow template `namespace`s so this isn't an argument for or against namespaces as classes.

Global Declarations vs Member Declarations

Currently a class member cannot be given C linkage. This must be allowed for modules. For example:

```
module M {
        extern "C" void f(int);
};
```

Similarly, `module` member templates must be allowed. For example:

```
module M {
        extern "C" void f(int);
        template<class T> class Y { /* ... */ };
        template<class T> f(T*) { /* ... */ };
};
```

The idea of wrapping a `namespace` declaration around a complete `.c` file does not have an equivalent for `module`s. That would have required the suspension of the rules for what initializers can appear in classes and of the rule that member functions defined within a class are inline by default.

Conclusions

A class is a type. A namespace is a more fundamental concept than classes. There is no significant benefit in merging the two concepts. There are several minor problems with treating a namespace as a class (all can be overcome). On balance, it is better to build the concept of a class on the concept of a namespace rather then the other way around.

16 Appendix C: Possible Further Extensions

Naturally, many extensions to the `namespace` and `using` proposal have been suggested. For example, several ideas have been along making namespaces more like classes or more like modules; these are considered in Appendix B. Here, I will just mention some variants of `using`.

A `using` declaration brings one or more names into the current namespace.

Renaming

However, what if I don't like the names chosen by the designer of the namespace. Maybe `using` should allow me to chose a synonym that is more to my taste. For example:

```
using g = A::f;
```

I think this would be a frill because C++ already has ways of expressing synonyms:

```
// no using

typedef A::String String;
int& m = A::n;
int (&g)() = A::f;
```

We don't have a general way of expressing synonyms independently of the type of the function, type, object, etc. named. If we want such a mechanism I suggest we look at ways of generalizing `typedef`. I don't plan to pursue this, though.

The reason for that is partly that I don't see a need for it, but primarily that I dislike chasing chains of aliases while maintaining code. If the name I see spelled `f` is really the `g` defined in the header which actually is described as `h` in the documentation and what is called `k` in your code then we have a problem. Naturally, this would be an extreme case, but not out of line with examples created by macro-aficionados. Every renaming requires understanding of a mapping for both users and tools.

The introduction of synonyms can be useful and occasionally close to essential. I don't see a need for extending the mechanism provided beyond the ability to introduce aliases for namespace names, though. Further features would simply encourage (mis)use of synonyms.

Exclusion

It was also suggested that we might like to include all names from a namespace except an explicitly named set. For example:

```
using A::(!f);  // all of A except f
using B::(!g);  // all of A except g

f(10);      // B::f
// ...
g(20);      // A::g
```

The idea is that if we find a clash arising from a plain

```
using namespace A;    // A has f and g
using namespace B;    // B has f and g

f(10);      // error: A::f or B::f?
// ...
g(20);      // error: A::g or B::g?
```

we simply modify the *using-directive* to exclude the ''offending'' names.

The problem with this solution is that it is indirect and brittle. We do not say what we want, but what we don't want. That poorly documents our intention and leaves room for the ambiguity to reappear if we add another namespace that again defines the excluded name. That scenario isn't unlikely for popular names such as `String`, `Boolean`, etc.

The solution is brittle because a adding a name in one of the namespaces may still break code or change the meaning of a program.

`prefer`

To compensate for the weaknesses of ''exclusion'' its logical compliment, an explicit expression of preference, was suggested:

```
using namespace A;    // A has f and g
using namespace B;    // B has f and g

prefer A; // meaning: in case of clashes use names from A
```

Unfortunately, this approach still suffers from being brittle and in a sense from being too powerful. Having ''preferred'' namespace `A` over namespace `B` adding a name to `A` can change the meaning of a program or make it stop compiling (just as in the case of ''exclusion''). The problem is that we have expressed an general preference for `A` rather than simply resolving existing ambiguities. This also means that preferring `A::f` and `B::f` isn't possible. One could remedy this by allowing preference for individual names to be expressed, but now the `prefer` mechanism is starting to elaborate exactly along the lines of `using`.

The proposal handles this example without added features through *using-declarations* in a namespace introduced to allow the disambiguation (see §5).

```
using namespace A;    // A has f and g
using namespace B;    // B has f and g

namespace {
        using A::g;     // hides other g
        using B::f;     // hides other f

        f(10);      // B::f
        // ...
        g(20);      // A::g
}
```

17 Appendix D: Overloading and Namespaces

One of the most vigorously debated issues about namespace was: Should functions overload across namespaces and if so should overloading somehow be restricted compared with ‘‘ordinary’’ overloading? This proposal suggests that overloading across namespaces be allowed according to the usual overloading rules. Where that is not desirable, I can either refrain from using *using-declaration*s and *using-directive*s, or wrap my functions in a separate namespace (§4). The reason for allowing overloading across namespaces are:

[1] Overloading together with ambiguity control is an important convenience (function with the same name and unrelated names doesn’t clash, so that I don’t have to take action to resolve such clashes; say, by renaming functions) and protection (calls that are ambiguous according to the usual C++ rules are caught).

[2] Overloading across `#include` header boundaries exists in current practice and overloading across namespace boundaries is essential to maintain current practices and upgrade current code.

[3] Use of overloaded of operators, especially current uses of `<<` becomes quite difficult if overloading across namespaces is not supported.

[4] The option *not* to overload is available in a convenient form (use explicit qualification or express a local preference by a *using-declaration*).

The Problem

The worry is that overloading across namespaces could become a maintenance problem. Consider:

```
namespace X {
        void f(A);
}

namespace Y {
        void f(B);
}
```

First note that the problem can only happen after someone applied a *using-directive* to both:

```
using namespace X;
using namespace Y;
```

This will cause overloading between names appearing in both `X` and `Y` even if the programmer is unaware that a name appears in both. A single *using-directive* combined with the use of genuinely global names can cause equivalent problems. Such problems currently occur as overloading across include files. Namespaces address such problems as long as multiple *using-directive*s are not used. I do not consider multiple *using-declaration*s a problem because, like ordinary declarations, each injects an explicitly specified name into a namespace.

The discussion was not over whether there was a problem, but about the magnitude of the problem. I think that the problems from *using-directive*s will be in the noise compared with problems from other sources. Others predict ‘‘horrendous maintenance problems.’’ All rest their cases on unreliable and subjective experience and logical arguments (also unreliable in practice). I think the real problem is that we cannot quantify the estimate of how often problems will occur and how hard such problems would be to detect and correct. Not all overload problems we now experience will re-surface as cross-namespace overloading problems.

Personally, I would minimize the use of multiple *using-directive*s and thus minimize the potential as well as the real problems. However, I don't see how I could realistically completely avoid such usage and I would not like to impose a rule against it upon all C++ programmers. That would be too much like imposing my own preferences and too much like trying to legislate morality.

Like others, I expect and recommend that purveyors of compilers and similar fundamental tools provide optional warnings against cross-namespace overloading resolution where more than one candidate resolution exist. People who is really paranoid (by ‘‘paranoid’’ I don't mean ‘‘stupid:’’ sometimes ‘‘they’’ really are out to get you) can work with such an option permanently enabled. However, I'm about as sure as I'll ever be that there are large application areas where prohibiting cross-namespace overloading resolution would be a mistake. That is, I'm for optional warnings, yet against a uniformly enforced prohibition.

Examples

Consider:

```
namespace X {
        class A { /* ... */ };
        ostream& operator<<(ostream&, const A&);
}

namespace Y {
        class B { /* ... */ };
        ostream& operator<<(ostream&, const B&);
}

using namespace X;
using namespace Y;

void f(A a, B b)
{
        cout<<a;        // X::operator<<
        cout<<b;        // Y::operator<<
}
```

I think this must be allowed, and therefore I consider a ‘‘no overloading across namespaces’’ unacceptable.

The nastiest problem is that of a call being ‘‘hijacked’’ by addition of a name in a previously unused namespace. Here, first is an example where only a single namespace is used:

```
namespace Q {
        class Quad {
                // ...
                Quad(double);
                operator double();
        };
        Quad sqrt(Quad);
}

using namespace Q;

void f(double d, Quad q)
{
        double sqd = sqrt(d);  // Q::sqrt(Quad(d))
        Quad sqq = sqrt(q);    // Q::sqrt(q)
}
```

Adding another namespace with a function that is a better match of a class and we quietly change the meaning of `f()`:

```
namespace std_math {            // expanded <math.h>
        double sqrt(double);
        // ...
}

using namespace std_math;

namespace Q {
        class Quad {
                // ...
                Quad(double);
                operator double();
        };
        Quad sqrt(Quad);
}
using namespace Q;

void f(double d, Quad q)
{
        double sqd = sqrt(d);   // std_math::sqrt()
                                // no longer Q::sqrt(Quad(d))
        Quad sqq = sqrt(q);
}
```

In some cases, the overload resolution will get ‘‘the right’’ answer relative to the programmer’s expectations; in some cases it will not. What ‘‘the right’’ answer relative to the programmer’s expectations is cannot be determined by a compiler. Often, problems are resolved when the programmer is a bit more specific about what the intent really is. In this particular case, the hijacking problem would have been solved had to programmer refrained from adding the implicit conversion from `Quad` to `double.`

One might say that a programmer ought to be alert to the possibility of changes of meaning when a new namespace is introduced, but realistically many will just add the namespace and hope for the best.

Further, the problem can occur in a slightly different form. Imagine (if you can) that originally `std_math` didn’t have a `sqrt()` function. Thus the original user code in the example above would look exactly as before and there would be no `std_math::sqrt()` to match `sqrt(d)`. If a new release of `std_math` adds `sqrt()` the meaning of the user’s program will quietly change.

There is no disagreement that this is a problem, only disagreement over its magnitude and over the effectiveness of techniques to minimize its occurrence.

Restriction

It has been suggested that overloading across namespaces should be disallowed. This would certainly solve the problem presented above, but at an a cost I consider unacceptable; it would prohibit the examples above and close the main transition path from the current (far worse) state of affairs to the use of namespaces.

Limiting overloading across namespaces to exact matches has been suggested. This is too restrictive because the addition of an unrelated function in another namespace could break existing code relying of a perfectly innocuous conversion. For example, in our `sqrt()` example the addition of complex `sqrt()` would break the `quad` example even if there were no conversions between `quad`s and `complex`s that could cause problems.

Limiting overloading across namespaces to operator overloading has been suggested. The counter-argument is that most programs that rely on operator overloading also rely on the overloading of a few significant names. The square root function is an example. Further, this too would block the migration path because people now rely on overloading across include files that would become separate namespaces. Limiting overloading across namespaces to operator overloading would also not solve the critical hijacking problem, only restrict its occurrence to operators.

Limiting overloading across namespaces to matches that involves only types and conversions from a single namespace has been suggested. Unfortunately, the `sqrt()` example shows that even exact matches can cause the hijacking problem. To me this indicates that only restrictions too Draconian to be acceptable can protect against the critical problem.

Conclusion

We need data to determine how serious the conjectured problems are. Unless new data appear indicating that the problems are serious I will consider them minor. Using the usual overloading rules is minimal (it introduces no new rules) and is therefore the simplest and least confusing solution. We need compilers and tools to provide an option (only) to warn against cross namespace overloading caused by *using-directive*s. We should recommend the use of *using-directive*s to be minimized. That is, to be used primarily for standard libraries and as a transition tool.

18 Appendix E: Name Injection

Some declarations require names to be injected into an enclosing scope. For example:

```
namespace A {
        void f(struct S* p) { ... }
        class X {
                friend class Y;
                friend int f();
        }
}
```

The usual rule is that such names are injected into the ‘‘nearest enclosing non-class scope.’’ I do not propose to change this (that’s why this is an appendix), but the decision wasn’t easy. This appendix explains the problem.

Namespaces and Injection

In the example above two alternatives are obvious:

[1] Inject the names into scope A.

[2] Inject the names into the global scope.

By choosing (1) we deem that a namespace isn’t a class-scope for the purposes of injection.

The primary purpose of namespaces is to ensure that names are only entered into a common scope through explicit programmer action. Thus I would hate to find that

```
namespace A {
        // anything here
}

namespace B {
        // anything here
}
```

could lead to a name clash. This is a strong argument that

```
namespace A {
        void f(struct S* p) { ... }
        class X {
                friend class Y;
                friend int f();
        }
}
```

should inject `S`, `Y`, and `f` into `A` and not into the global scope. This rule aids comprehension and the building of systems for manipulation of larger units of code.

Consider a more complicated case:

```
namespace A {
        struct U {
                struct V *v;     // "V" injected into A::
                friend void f(); // "f" injected into A::
        };
}
```

By a simple transformation this becomes

```
namespace A {
        struct U;       // forward declaration of U
}

struct A::U {
        struct V *v;     // "V" injected into A::
        friend void f(); // "f" injected into A::
};
```

In this last case, one could seriously consider injecting `V` and `f` in the global scope. After all, we are defining a struct and the global scope is certainly enclosing. However, if we inject into the global scope the namespace ‘‘leaks’’ names into the global scope and in-class and out-of-class definition gives different results. Thus, for `A::U` the nearest enclosing non-class scope is `A` and not the global scope.

This rule isn’t without negative aspects. If we implicitly inject into the namespace we cannot determine the set of names in a namespace by listing explicitly declared the members of a namespace. This was an arguments against injection into class scopes. However, one of the fundamental differences between classes and namespaces is exactly that a class is has a fixed and known set of members, whereas a namespace is open so that we can add names at any point.

Why don’t we ban injection? Because, if we ban injection we cannot wrap arbitrary declarations into a namespace. For example:

```
class X {
        friend void f();  // f() is global
};

void f() { /* ... */ }
```

works so

```
namespace A {
        class X {
                friend void f();  // f() is A::f()
        };

        void f() { /* ... */ }
}
```

ought to work also. Note that this can be expected to be a common case because of the popularity of defining operators as friend functions. A further decomposition yields:

```
namespace A {
        class X;
        void f();
}

class A::X {
        friend void f();  // f() is A::f()
};

void A::f() { /* ... */ }
```

From these examples we conclude that a friend function defined within a namespace is assumed to be a member of that namespace. If we want a friend to be global we must declare it in the global scope:

```
void f();

namespace A {
        class X {
                friend void f();  // f() is global
        };
}
```

Namespaces and `extern`

A similar problem occurs for linkage and have a similar resolution for the same reasons. Consider:

```
namespace A {
        void g()
        {
                extern void f();  // global or in A?
        }
}
```

Is `f()` assumed to be in the global namespace or in `A`? Again we consider a program being transformed from traditional style into using namespaces:

```
void g()
{
        extern void f();
}

void f() { /* ... */ }
```

This becomes:

```
namespace A {
        void g()
        {
                extern void f();
        }

        void f() { /* ... */ }
}
```

If `f()` hadn't been part of the program fragment that became `A` but a completely different program fragment (say a standard library) if should have been declared by including the relevant header. Again, we can resolve the problem by adding a global declaration:

```
void f();

namespace A {
        void g()
        {
                extern void f();
        }

        void f() { /* ... */ }
}
```

This exactly parallels the name injection solution. Note that in both cases the solution was to choose locality as the default and allow resolution to global scope/linkage by a declaration that didn't actually modify the functions involved.