Doc No: SC22/WG21/N1344
J16/02-0002

Date: 1 March 2002

Project: JTC1.22.32

Reply to: Herb Sutter
Microsoft Corporation
1 Microsoft Way
Redmond WA USA 98052-6399
Fax: +1-928-438-4456
Email: hsutter@acm.org

Namespaces and Library Versioning

Now that work on the next version of the C++ standard library is underway, there are some basic logistical questions that need to be answered about the C++0x facilities:

- What namespace(s) are they going to go in?
- Do we want to support source compatibility (existing C++ programs using the C++98 standard library continue to work with unchanged meaning)? This is almost certainly necessary.
- Do we want to support binary compatibility (vendors can ship a C++0x standard library that is link-compatible with code written to their C++98 standard library)? This is less obviously necessary, although some users and some vendors will consider it essential.

These questions are thornier than they look. The choices that are made for the C++ standard library ought to be exemplary, showing library writers in general how library versioning ought to be done in C++. And, anyway, that's one of the big things namespaces were supposed to be good for, right? So don't we have the tools to do a better job than namespace-less languages? Alas, it's not quite that simple, as we shall see.

Source and Binary Compatibility

First, consider one of the most basic issues facing any library vendor contemplating a new release: whether to support source compatibility, binary compatibility, both, or neither. The answer may be different for different parts of a library; for example, the vendor may choose to maintain source compatibility in general but give it up as a tradeoff in the case of a particular facility.

Source compatibility loosely means that the user may have to recompile existing code, but the code will still work and its meaning won't change. If you want to maintain source compatibility with older versions of a library, don't change the meanings of any existing names or functions. Pure syntactic extensions that that won't alter the meaning of existing code are okay.

What's okay for source compatibility:

- Adding new facilities (classes or functions) with different names is okay. They won't conflict with anything else that users may be doing with the library, and if some user just happened to already have something lying around with that same new name, oh well, it's bad luck.
- Adding new defaulted parameters to existing function signatures is okay, as long as when existing code is recompiled it will use the defaulted parameters and run with the same semantics as before (but see a caveat below under "what's not okay").
- Similarly, adding new defaulted template parameters to existing templates in a way that preserves the original semantics is okay (also with caveats, described next).

What's not okay for source compatibility:

- Adding new overloads of existing function names is often not okay. For example, in the presence of conversions, some existing code may end up calling the wrong function (or none at all if it has become ambiguous). In rarer cases, if existing code attempts to take the address of the function, adding new overloads will make that ambiguous.
- Adding new defaulted parameters to a function, although generally okay as noted above, could break existing code that tries to take the address of the function and might depend on exactly its original signature; for example:

```
// Example 1-1
//
typedef int (*PF)(int);

int f( int ); // original library function
PF pf = &f; // okay

int f( int, float = 0.0 ); // changed library function
PF pf = &f; // error: no longer okay
```

- Similarly, adding new defaulted template parameters to a template, although generally okay, could break existing code that uses template template parameters and thus relies on the exact original number of parameters:

```
// Example 1-2
//
template<template<typename U> class V>
  class Y { V<int> v_; };

template<typename T>
  class X { }; // original library template
Y<X> yx; // okay

template<typename T, typename U = int>
  class X { }; // changed library template
Y<X> yx; // error: no longer okay
```

Next, *binary compatibility* loosely means that the user is not required to recompile any existing code, but only relink it with the new library, and that the link will still work and the resulting executable's meaning won't change. If you want to maintain binary compatibility with older versions of a library, be careful that you don't change any existing names or function signatures. These participate in name mangling, and absent some really

Herculean efforts this won't work.

What's okay for binary compatibility:

- Adding new facilities (classes or functions) with different names is okay.
- Adding new overloads of existing function names is okay because these won't cause link errors.

What's not okay for binary compatibility:

- Adding new defaulted parameters to existing function signatures will cause link incompatibilities.
- Adding new defaulted template parameters to existing templates will cause link incompatibilities.
- Perhaps the biggest issue in terms of this article's discussion: *Changing namespaces* will likewise break link compatibility, which is one of the biggest problems with Option 2 "Move out, but stay in touch" below.

The above lists, summarized as a Venn diagram in Figure 1, are not meant to be exhaustive. There are many more "what's okay" and "what's not okay" issues for source and binary compatibility, some of them platform-dependent. For example, changing entry points: moving a function's ordinal entry point number in a DLL or changing a member function's vtable slot are binary-incompatible even if the function itself is otherwise unchanged.

This is a sufficient summary of major source and binary incompatibility issues for the purposes of this article. You should get the sense that, in general, binary compatibility is more difficult to maintain than source compatibility, but that they are different if overlapping problems and there are cases where binary compatibility is easier to maintain than source compatibility.

Having those issues well in mind, now, let's get back to the bigger issue: Version control for a C++ library, specifically the C++ standard library.

Finding a Home: The Basic Choices

When it comes to finding a home for the next-generation C++0x standard library, the question boils down to two basic choices:

Option 1: Live with your parents. It's cheap, and it's perfectly fine if you and your folks can get along together well. Taking this route for the C++0x standard library means dumping everything in the same place where the C++98 library already lives, namely namespace `std`. This is one obvious possibility, because this is precisely what libraries in namespace-less environments, such as the C99 standard library, have to do. But for C++ it may smack a bit of inelegance, mightn't it, because weren't those newfangled namespaces supposed to be a useful tool for versioning libraries? Hence the second option:

Option 2: Move out, but stay in touch. This can be more flexible, but it's more work, and depending on your self-restraint and maturity (or lack thereof) it can also get you into trouble if you're not careful. Taking this approach would put the C++98 and the C++0x libraries in different namespaces. Add magic to make life easier for backward compatibility with existing code and migration to use the new facilities (there are many spices and seasonings to choose from here, so be creative), stir well, and hope for the best.

Option 1: Live With Your Parents In ::std

This option is the most straightforward. The issues are pretty much as described above, with few new wrinkles.

Option 2: Move Out Into Another Namespace, but Keep In Touch with ::std

The scenario: The entire new C++0x standard library, including parts it shares with the original C++98 standard library, has moved out of its parents' place (`::std`) and lives on its own in a new namespace (for the sake of discussion, let's call it `::std2`). The original C++98 standard library continues to live in namespace `::std`, blissfully oblivious to the wild parties going on across town night after night in `std2`'s bachelor pad with its new additions (such as those newfangled hash-based containers) and other differences (maybe `std2` got bored of always tripping over the `vector<bool>` specialization lying around on the floor and tossed it out with last week's trash).

This works reasonably well in the simple cases:

Using existing code with the old standard library: Existing code continues to work unbroken. There are no changes in meaning, because of course `std` is still the good old `std` it always was.

Switching wholesale to the new standard library: Using the new standard library as a drop-in replacement in existing programs is as simple as globally replacing `std::` with `std2::` and changing "`using namespace std;`" to "`using namespace std2;`". (For a discussion of why using-directives are not evil, see Item 40 in [1].) The only changes in meaning would be to facilities that exist in both versions of the library and that the committee explicitly decided to change; these are expected to be few and well-documented.

There are three areas where this approach works less well. The first was already noted above: Changing names breaks link compatibility, and some of the techniques we'll discuss in a moment rely on changing the namespace of residence for facilities currently resident in `std`.

The other two areas are worth delving into in some detail. One area is visible to the user, and one is visible to the library writer. Let's tackle them in that order.

Option 2, First Problem: Using Existing Code with the New Standard Library

User code that wants to use some of the new standard library can do it by asking for `std2::`, but that can still be annoying. Consider the following existing code:

```
// Example 2-1:
// PROBLEM: An innocent happy declaration
// in some existing header file.
//
int f( std::vector<int>& );

// Sample code that uses it does not play
// well with the new library because of std2.
//
std::vector<int> int v1;
f( v1 ); // ok

std2::vector<int> int v2;
f( v2 ); // error, f() only wants a std::vector
```

This is a real issue. "Ah, but that's easy to fix," some may say, "just don't write `std::` in the function declaration, just write a using declaration or using directive near the top of `f()`'s header and you'll only have to change it in one place in each header." But that's not right: you *should* indeed write an explicit `std::` in the function declaration, because it turns out there are good reasons to never write namespace using declarations or directives in header files, ever. Always use explicit namespace qualifications in function declarations like `f()`'s

(see again Item 40 in [1]).

Now, notice that the problem does not occur if `f()` is already a template like so:

```
// Example 2-2: (BAD)
// An impractical non-solution.
//
template<typename T>
int f( T& );

// Sample code that uses it is now okay:
//
std::vector<int> int v1;
f( v1 ); // ok, uses f<std::vector<int> >

std2::vector<int> int v2;
f( v2 ); // ok, uses f<std2::vector<int> >
```

Although the above code is perfectly legal, it's not a 'solution' to the problem. I think it's untenable that we tell users that if they write their own function that happens to mention a standard facility as a parameter type, they should make their function a template. (Ditto for their own classes.) That's just foolish. In the above case, there's probably no reason why `f()` should be templated. Dispensing advice to make all such constructs into templates would mean recommending lots of user templates, never mind a renewed round of vigorous accusations that C++ is complicated. Let's not go there. We definitely would like functions like "`int f( std::vector<int>& )`" to keep working with the new standard library `vector`, whether `vector` is changed in C++0x or not.

Option 2, Second Problem: How To Implement and Maintain Separate C++98 and C++0x Library Namespaces

I usually write about C++ from the point of view of C++ programmers in the trenches who have to use this stuff. After all, that's the viewpoint of most of us. For just a moment, though, come along and let's pretend that we're standard library implementors, and see how we might deal with a requirement to implement the full C++98 standard library in namespace `std`, and the full C++0x standard library (whatever that turns out to be, but probably with major additions but also with most common things completely unchanged) in some other namespace `std2`.

The main question is what to do about all the common things that are unchanged from C++98 to C++0x. We'd sure like not to have to write them all twice, once in `std` and once in `std2`! Here follow some major options, and why they're problematic.

Option 2(a): Code-Pasting Via #include <impl>

The idea in this "code-pasting" option is to put the implementation of some facility that hasn't changed (let's pick on `vector` again) into a common implementation file, then brute-force `#include` that in both namespaces:

```
// Example 3
//

//----------------------------------------
```

```
// __vector_impl.h: common stuff
namespace __STDNAMESPACE
{
  template< /*...*/ >
    class vector { /*...*/ };
}

//-----------------------------------------
// C++98 header
#define __STDNAMESPACE std
#include <__vector_impl.h>
#undef __STDNAMESPACE

//-----------------------------------------
// C++0x header
#define __STDNAMESPACE std2
#include <__vector_impl.h>
#undef __STDNAMESPACE
```

This avoids code duplication, which is good.

A big drawback, however, is that it does not give a true shared implementation: If the same program uses `std::vector<int>` and `std2::vector<int>`, we may be in for some link-time bloat for non-inlined functions because there are two implementations, which happen to be identical except that because they're in different namespaces their mangled names vary and so most linkers wouldn't eliminate one. Perhaps an imaginary omniscient linker could notice that the generated code for both was identical *and* that the parameter lists were basically identical and fold the two copies, but noticing that the parameter lists were really identical would be tough when, for example, the two versions of the default constructor would still take different types of parameters (for example, `std::allocator<int>` vs. `std2::allocator<int>`). It's probably not practical to imagine omniscient linkers. Besides, many commercial linkers are still only C-aware, not C++-aware even for basic things, much less such esoterica like this.

Now, although the naïve cut-and-paste brute-force approach above wouldn't share implementation well, we could take it a step further:

Option 2(b): Wrapper Around an __impl

The idea here is to wrap both `std::vector` and `std2::vector` around a common implementation, say `__myob::__vector_impl`. It could look something like this:

```
// Example 4
//

//-----------------------------------------
// __vector_impl.h: common stuff
namespace __myob
{
  template< /*...*/ >
    class __vector_impl { /*...*/ };
}

namespace __STDNAMESPACE
{
```

```
  template< /*...*/ > class vector
  {
    // implemented in terms of
    // __myob::__vector_impl
  };
}

//----------------------------------------
// C++98 header
#define __STDNAMESPACE std
#include <__vector_impl.h>
#undef __STDNAMESPACE

//----------------------------------------
// C++0x header
#define __STDNAMESPACE std2
#include <__vector_impl.h>
#undef __STDNAMESPACE
```

Because the visible `std::vector` and `std2::vector` are just passthroughs, they likely won't incur similar link-time code duplication even if `std::vector<int>` and `std2::vector<int>` are used in the same program, because all their functions are typically one-line `inline` forwarding functions. The single shared `__vector_impl` does all the real work, and this time there truly is only one of it. We don't need to imagine an omniscient linker to strip duplicates, because the library implementor prevented duplicates from occurring.

Option 2(c): Use "using"

One might think that a more C++-ish way of hoisting a facility into two namespaces would be to use a using-declaration. One would be almost right. The idea here could look something like this:

```
// Example 5-1: Alternative 1,
// real declaration lives elsewhere
//
namespace std
{
  using __myob::vector;
}

namespace std2
{
  using __myob::vector;
}
```

or like this:

```
// Example 5-2: Alternative 2,
// real declaration lives in std
//
namespace std2
{
  using std::vector;
}
```

or even like this:

```
// Example 5-3: Alternative 3,
// real declaration lives in std2
//
namespace std
{
  using std2::vector;
}
```

Alas, this also has problems. (Surprise.) In this case, the problems stem from the fact that an entity pulled in via a using declaration doesn't have all the same status and perks as the real declaration.

Here's one of the problems: Users are allowed to specialize standard library templates on their own user-defined types. That's a perfectly legal C++98 technique today. But, if the above alternatives were allowed and you wanted to specialize `vector`, where would you specialize it? The specialization must reside in the same namespace as the original template, after all, and the user might not be able to tell where the original template lives, and it's especially bad if that answer changes from C++98.

Consider the following currently-legal code:

```
// This is legal today in C++98:
//
class MyClass { };

namespace std
{
  template< > class vector<MyClass> { };
}
```

If we were to adopt Alternative 3 above, then code could still refer to `std::vector<int>`, but the above attempt at partial specialization would break:

```
// Alternative 3 again: Why it's unworkable
//
namespace std
{
  using std2::vector;
}

// Used to be legal in C++98, but would break
// if the above were adopted:
//
class MyClass { };

namespace std
{
  template< > class vector<MyClass> { };
    // error, base template is in std2, not std
}
```

Indeed, we'll now see that the fourth and similar option has the same problem:

Option 2(d): Use an alias

Namespace aliases are a little-known feature of C++, if the amount of press they get is any indication. Most

authors don't talk about them much. What they do is really simple, and here it is:

```
// Example 6: Namespace aliases
//
namespace SomeLongNameThatsAnnoyingToSpellOut
{
  class Thing { };
}

namespace That = SomeLongNameThatsAnnoyingToSpellOut;

That::Thing t;
```

That is, a namespace alias is just another name for a namespace. Alas, what you can't do is reopen the namespace using the alias:

```
namespace X = Y;

namespace X // error, can't reopen X, it's an alias
{
  // ... never get here ...
}
```

Why would we care about this? Because, in particular, it might be nice to rename namespace `std` to `std1`, say, and put the C++98 standard library there; and then create a new `std2` namespace and put the C++0x standard library there; and then just use `std` as an alias for the real namespace name:

```
namespace std = std2; // wouldn't this be nice?
```

But then you're back into I-can-refer-to-things-using-that-name-but-I-can't-specialize-templates-using-that-name land:

```
// Used to be legal in C++98, but would break
// if the above were adopted:
//
namespace std // error, can't reopen std... rats
{
  template< > class vector<MyClass> { };
}
```

The user would have to know about the implementation technique and instead reopen namespace `std2`. And so, alas, when it comes to namespace aliases for library versioning, saying "`=`" is not quite enough.

Summary

There are some secondary issues I've not addressed in the this discussion. Some of the issues I did address had some details slightly glossed for ease of presentation. But the fundamental issues and problems are valid ones, and seeing this should give you a flavor for the issues involved in trying to use C++98 namespaces to manage version change in new library releases. The standard library itself is the case in point, but the discussion applies equally well to any third-party vendor library, for all libraries face these versioning challenges.

There are additional options that are so bad I haven't even talked about them. For example, one 'anti-solution' to Example 2-1 would be to inherit `std2::vector` from `std::vector`. This is an absolutely horrid idea. For one thing, it would add overhead to `std::vector`, because then `std::vector` ought to be polymorphic solely to make a compatibility hack work. For another, that same addition of polymorphism would

change `std::vector`'s design in a way that gratuitously breaks binary compatibility. For a third, it's against the spirit of generic programming. For a fourth, if a fourth be needed, it's just plain ornery wrongheaded, probably fattening, and known to cause cancer in laboratory animals.

When Tom Cargill published his seminal 1994 article "Exception Handling: A False Sense of Security," he showed that we as a community didn't really yet know how to write exception-safe code. What was interesting about that article was that he ended his analysis, not by saying "and here's how to do it," but rather by saying "don't think that the issues I've listed are all the issues because there are issues I know about that I haven't discussed here, I don't even know if I know all the issues, I don't think anyone else does either, so I encourage someone in the C++ community to write an article showing how it's done." It took three years for such an article to appear (that material, since greatly expanded, is now the Exception Safety section of [2]).

I'd like to end this article on a similar Cargillesque note: I claim that, due to the above issues and others, we as a community don't really yet know how to use namespaces to effectively version library releases, and at this point we can't do better than just keep everything in namespace `std` forever. There are issues I haven't discussed here, I don't even know if I know all the issues, I don't think anyone else does either, and so I encourage someone in the C++ community to write an article showing how it's done. (There is a group of interested people inside the committee's library working group discussing this very problem, which group I (was) volunteered to coordinate, and if an answer is forthcoming it will likely be from someone in that group. That doesn't mean that other experts shouldn't try, though.)

Even if a complete solution is possible, I suspect that it will almost certainly require at least minor changes to the core language namespace feature. In the meantime, don't be surprised if the eventual C++0x standard library, extensions and all, still ends up living at home in namespace `std`. Sometimes the outside rent is just too costly, and it pays to live at home.

References

[1] Herb Sutter. *More Exceptional C++* (Addison-Wesley, 2002) , http://www.gotw.ca/publications/mxc++.htm.

[2] Herb Sutter. *Exceptional C++* (Addison-Wesley, 2000), http://www.gotw.ca/publications/xc++.htm.

[3] Tom Cargill. "Exception Handling: A False Sense of Security" (*C++ Report*, 9(6), November-December 1994).

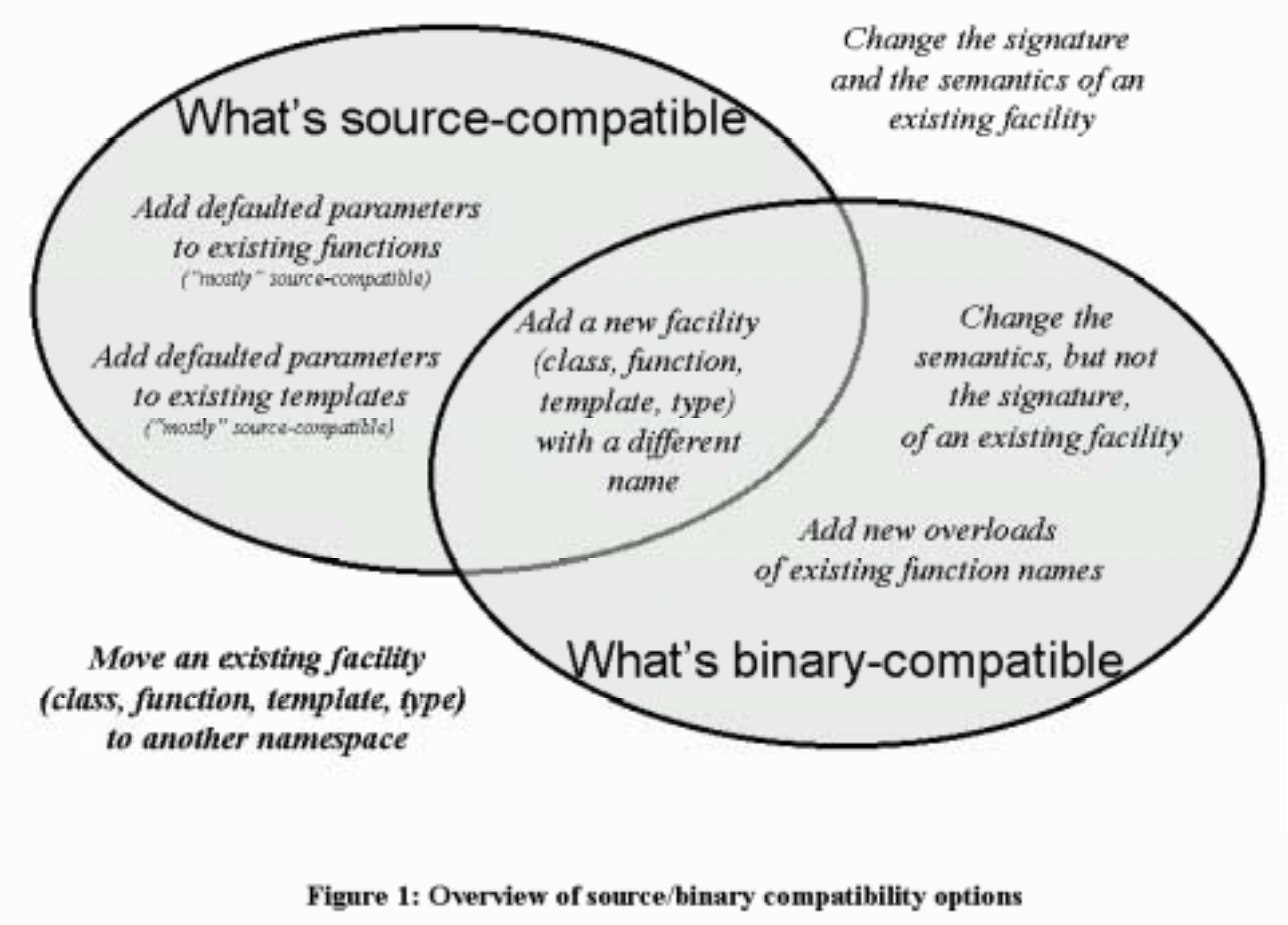

Figure 1: Overview of source/binary compatibility options