Accredited Standards Committee*
X3, INFORMATION PROCESSING SYSTEMS

Doc No: X3J16/95−0045
WG21/N0645
Date: 31 January 1995
Project: Programming Language C++
Reply to: Andrew Koenig
AT&T Bell Laboratories
PO Box 636
600 Mountain Avenue
Room 2C−306
Murray Hill, NJ 07974 USA
ark@research.att.com

Reconciling overloaded operators with namespaces

This note identifies a critical problem and outlines a partial solution.

To see the problem, consider the following example:

```
#include <iostream>

main()
{
      std::cout << "Hello world\n";
}
```

The problem is how to locate the definition of `<<`. In the example shown, it is defined as a member of the class of `std::cout`, so there is no problem. However, suppose we change the example slightly:

```
#include <iostream>
#include <string>

main()
{
      std::string s = "Hello world\n";
      std::cout << s;
}
```

Now, `<<` is not a member of the class of `std::cout`. Indeed, its definition is not visible at all from the point of use, so this program is ill-formed.

How do we make it work? There are several possibilities, all of them ugly. First, we might make the scoping explicit:

```
      std::operator<<(std::cout, s);
```

If we require this, we will deserve all the ridicule we get. Instead, we might insert

```
      using std::operator<<;
```

where it will be visible from the use of `<<`. This defeats the whole point of overloading, which is to make it possible to avoid having to say where functions or operators are defined.

---

* *Operating under the procedures of the American National Standards Institute (ANSI)*
Standards Secretariat: CBEMA, 1250 Eye Street NW, Suite 200, Washington DC 20005

I would hate to have to teach novices either of these alternatives.

Instead, I think the right solution is to recognize that overloaded operators already have unusual properties with respect to scopes and say that an overloaded operator is looked up not only the scope(s) that are directly visible from where it is used but also in the scope(s) that contain the definition(s) of the type(s) of its operand(s).

This would legalize the string example above. It would also solve another hotly debated problem:

```
class X { /* ... */ };
bool operator==(const X&, const X&);

main()
{
	X x1, x2;

	// ...

	if (x1 != x2) { /* ... */ }
}
```

The question is whether or not the expression `x1 != x2` should pick up the `!=` template defined as part of STL. With a rule of the kind I am proposing, the answer would be ''no;'' and I think that is the answer that most users want. Moreover, if we were to move the STL relational templates into their own namespace, say `std::relations`, then the author of class `X` could potentially make `!=` available by some legerdemain like the following:

```
namespace Foo {
	using std::relations;
	class X { /* ... */ };
	bool operator==(const X& const X&);
};
```

Then if the rules were just right, a user of class `X` might be able to say

```
main()
{
	using Foo::X;

	X x1, x2;

	if (x1 != x2) { /* ... */ }
}
```

and have it work automagically.

This is admittedly sketchy. Its intent is to identify a problem that I think must be solved before we can proceed to CD approval and public comment and write down my present thinking about a likely solution. I hope that someone will be able to contribute to the analysis in the Austin meeting.