Doc No: X3J16/92-0008
WG21/N0086
Date: January 31, 1992
Reply to: Roland Hartinger

A Proposal solving the Name Space Pollution Problem in C++

Volker Bauche
Roland Hartinger
Erwin Unruh

Siemens Nixdorf Informationssysteme AG
Department of C/C++ Compiler Development STM SD224
Otto - Hahn - Ring 6
W-8000 Munich 83
Germany
email: unido!sinix!athen!d015s000!hartinger

1 Introduction

An important unsolved problem in C++ is that it doesn't support a mechanism to avoid name conflicts while using libraries from different independent vendors.
Such libraries provide type names, function and variable names which can interfere with each another when used together in an application program.
Therefore the language should extended by an appropriate mechanism. A solution to the problem is described precisely below.

2 Proposal

Currently C++ handles four different kinds of scope, namely: Local, Function, File and Class.
We suggest therefore the introduction of another kind of scope , the

'Named Scope'.

A named scope can be introduced by using the already existing '::' scope resolution operator. For example,

```
::NewScope:: {
    class X {
        // ...
    };
}
```

Offene Fragen:
a) Scope declaration
b) should it be allowed, to nest named scopes?
c) overloading only on operator functions

hides the class X in the scope ::NewScope:: from possible other class X declared in a file scope.

Since no name ambiguity arises during reference to a name, it is not necessary to specify the scope name explicitly. For example,

```
// windows.h supplied with the window library by SNI
::SNIwindow:: {
   class window {    // window declaration
   };
}

::My:: {             // function in another named scope
   void f(window&); // ::SNIwindow:: is taken
}

// using windows from SNI in the following program
#include <windows.h>

f(window&) {
   window wd1;       // the compiler associates window with
                     // ::SNIwindow::window

   f(wd1);           // ok: global f hides named f,
                     // consequently
                     // this is a recursive call of f()

   ::My::f(wd1); // ok
}
```

3 Goal of the Named Scope

The goal of the introduction of Named Scope is to prevent name collision when using libraries and their headers which can contain possible conflicting type, function, and variable names.

The following example shows a class window which is encapsulated in the named scope SNIwindow. One can see that scope name is delimited by the :: operator followed by a scope block which is syntactically the same as a declaration-list.

```
::SNIwindow:: {          // new scope starts here
        class window {
        // ...
        public:
                  window& open();
                  void draw(int, int);
                  friend int f1();
        };
        //...
}                         // and ends here
```

An object of that class can be created like any other object of a type which is in the same file or local scope as far as no other class of that name is in effect. For example:

```
window wd1;     // equivalent to ::SNIwindow::window wd1;
```

If another window class is in effect, for example

```
class window { // another window class
               // in the actual (say file) scope
};
```

the SNIwindow object must be defined explicitly with its scope name, like :

```
::SNIwindow::window wd1;
```

4 Changes in the Language

4.1 Complete the Grammar

The following changes in the language are necessary to support the named scope. The changes are based on the grammar of chapter 17, Appendix A in the "Working Paper for Draft Proposed American National Standard for Information Systems - Programming Language C++", DOC NO: X3J16/ 91-0009, Date: February 11, 1991.

:
:

17.2 Expressions

:
:

allocation-expression:

::$_{opt}$ `new` *placement*$_{opt}$ *new-type-name new-initializer*$_{opt}$
::$_{opt}$ `new` *placement*$_{opt}$ (*type-name*) *new-initializer*$_{opt}$
scope-name `new` *placement*$_{opt}$ *new-type-name new-initializer*$_{opt}$ //to add
scope-name `new` *placement*$_{opt}$ (*type-name*) *new-initializer*$_{opt}$ //to add

:
:

deallocation-expression:

::$_{opt}$ `delete` *cast-expression*
::$_{opt}$ `delete []` *cast-expression*
scope-name `delete` *cast-expression* //to add
scope-name `delete []` *cast-expression* //to add

:
:

primary-expression:
literal

`this`
`::` *identifier*
scope-name identifier //to add
`::` *operator-function-name*
scope-name operator-function-name //to add
`::` *qualified-name*
scope-name qualified-name //to add
`(` *expression* `)`
name

⋮

17.3 Declarations

declaration:
scope-name `{` *declaration-list* `}` //to add
decl-specifiers$_{opt}$ *declarator-list*$_{opt}$ `;`
asm-declaration
function-declaration
template-declaration
linkage-specification

⋮

simple-type-name:
complete-class-name
complete-type-name //to change
`char`
`short`
`int`
`long`
`signed`
`unsigned`
`float`
`double`
`void`

⋮

complete-type-name: //to add
qualified-type-name //to add
scope-name qualified-type-name //to add

⋮

complete-class-name:
qualified-class-name
`::` *qualified-class-name*
scope-name qualified-class-name //to add

scope-name: //to add
`::` *identifier* `::` //to add

5 Scopes

5.1 The Lookup Rules

A full qualified name is searched only in the named scope, which is given in the qualification. A name without qualification is searched in the following scopes:

1. in the local block-scopes, if it appears in a function
2. in the class scope, if it appears in a class definition or a member function definition

3a. in this named scope, if it appears inside the definition of a named scope or

3b. in the global scope, if it appears outside any definition of a named scope

4. in every named scope and the global scope; if it appears in two scopes, the use is ambiguous.

5.2 Restrictions

It is forbidden to declare a global class with the same name than a named scope!

Consider the following piece of code:

```
::A:: {
    int x;                          // No. 1
}
struct A {
    static int x;                   // No. 2
};
void f()    \

    struct A { static int x; } // No. 3
    int i;

    i = A::x;                       // using No. 3

    i = ::A::x;                     // using No. 1 through
                                    // "named scope with name"
\\                                  // or using No. 2 through
                                    // "global class with member"
                                    // cannot decided here !
```

5.3 Special Cases

5.3.1 Nested named scopes

The names of a declaration are local to its named scope. The names inside a named-scope-declaration can be referenced as names without scope-name as long as there is no ambiguity recognized. An name clash can be solved by using the scope-name explicitly to specify the desired name.

Names scopes can be nested, for example

```
::X:: {
    class A;
    ::Y:: {
        class A;
    }
}
```

But to keep name-scope rules simple, the classes A in the above declaration getting the names,

```
::X::A
::Y::A
```

however not the one, one would commonly expect as ::X::Y::A.
This can be easily seen also from the scope-name syntax, because it starts at :: which means 'beginning in the global scope' and not starting new scope relative to the actual one.

5.3.2 Operator Functions

Normally the named scope resolution for function names is completely provided without any help of the overload resolution mechanism.

For example,

```
::A:: {
    extern int f1(int);
}

::B:: {
    extern int f1(double);
}

foo() {
    int i;

    f1(i);              //error: though it is possible
                        // to decide for ::A::f1()
}
```

This is restricted also to simplify the scope resolution problem in cases, where it is much harder to match the names.

On the other hand, consider the following example,

```
::A:: {
    operator <<(stream&, int);
}

::B:: {
    operator <<(stack&, int);
```

```
}

stack st;
stream cin;

cin ::A::<< 1;              // error: strange and not possible

::A::operator <<(cin, 1); // possible but not convenient
                          // because the advantage of
                          // operator overloading is lost.

st << 1;                  // uses operator << from Scope
                          // ::B:: because of a match of
                          // that param's
```

The reason for applying overload resolution only on operator functions is that they are 'restricted' on parameter types. They need always one class parameter. This makes the name resolution easy and clear.

5.3.3 External Declaration and Named Scope

It is legal to have an external declaration inside a named scope, for example,

```
::SNImath:: {
    extern "C" {
        int errno;
        double sin(double);
    }
}
```

The errno and sin() are part of the enclosed scope. But there is no impact on their external names. The linkage specification gives the direction of the real external name representation.

5.3.4 Named scopes can be added

Suppose you want to write a program, then it might be helpfull to use the headers of that library in the following fashion,

h1.h
```
::Xlib:: {
    class A { // ...
    };
    // ...
}
```

h2.h
```
::Xlib:: {
    class B { // ...
    };
    // ...
```

```
}
```

user.c

```
#include "h1.h"
#include "h2.h"

main() {      // ... using both headers of Xlib - Library
    A a;      // both classes A and B share scope ::Xlib::
    B b;
    // ...
}
```

6 Acknowledgment

Special thanks to Dag Brueck, Steven J. Dovich and Bjarne Stroustrup for their help discussing and reviewing this document.