X3J16/91-0041

# Name Space Pollution: A Proposal

Version 1: Feb 4, 1991
Version 2: March 9, 1991

Keith Rowe
Microsoft Corp.

## 1.0 The Problem

One of the goals of C++ is to encourage the re-use of code. As this practice becomes more common, more and more of our code will be pieced together from previously written components.

The problem is that groups of classes from different authors may inadvertently select identical symbol names for classes, enumerations, member functions and objects that may act very differently. The likelihood of such a name clash grows exponentially with the number of authors contributing to the new work.

This problem will affect both those who try to generate classes for re-use and those who try to combine the classes again.

### 1.1 An Example

Vendor A creates a hierarchy of classes for window management and decides to model a list box with a class called **List**.

Vendor B creates a second hierarchy of classes to handle collections and decides to model sequentially ordered collection with a class called **List**.

Even though these two sets of classes may complement each other well, the re-use of the name **List** will make it very difficult to combine these classes or even to use them in the same application. If the consumer of the classes doesn't have access to the source code for the classes, it may even be impossible.

What we need is a convention, mechanism or combination of the two that allows authors to prevent symbol collisions between their groups of classes. This should be easy to use and minimize the inconvenience to the consumer of the classes.

## 2.0 Rejected Solutions

One approach would be to suggest a convention where all vendors use a unique prefix for all their symbol names. The X Windows and Motif libraries are examples of this solution.

While this is a simple solution that requires no change in the language, it does require awkward symbol names. Will users of your class always want to type MS before every symbol? Also, there is no guarantee that two vendors may not accidentally choose identical prefixes. This will be compounded by the natural drive to keep prefixes short. For example, what keeps Michigan State and Microsoft from both choosing to append MS to all their symbols?

## 3.0 "Bundle" and "Use"

A **bundle** is a group of declarations that share a common named scope, much like nested classes. Bundles are neither types nor classes, and thus, instances of a bundle cannot be created. Bundles can be nested within other bundles. Individual declarations within a bundle can be accessed using the existing notation for nested classes.

The syntax for declaring a bundle is:

> **bundle** *identifier* {
>
> *declaration-list*
>
> }

For example:

```
bundle Microsoft {

     class Window {          // class declaration
          // ...
     };

     int windowCount;        // type declaration

     void tallyWindows(Window w)  // regular function
     {
          //...
     };
};


main ()
{
     Microsoft::Window myWindow;

     Microsoft::windowCount = 1;

     Microsoft::tallyWindows(myWindow);
}
```

This provides authors with a convenient mechanism for uniquely marking their class libraries. Since full identifiers are used to name bundles, it is convenient use long strings that cannot conflict (company names or modified internet addresses, for example). Since bundles can be nested, various groups within a larger project can also protect their name spaces (eg. the windowing team can use **Microsoft::Windows** and the spreadsheet team can use **Microsoft::Excel**). The mechanism of using the '::' operator to distinguish scopes is already used in C++.

The chief drawback of this addition is the long symbol identifiers that the consumer of the class library must now use.

This is solved with the use keyword. The statement:

use *bundle-identifier*;

can be introduced at any scope in a function definition or at global scope. From that point on in the current scope, any identifier that cannot be found in the symbol table is tried again with the bundle identifier prefixed to it. Multiple use statements can be active and the bundles are checked in order from the most recently declared use statement to the least recently declared.

For example:

```
bundle Microsoft {

     // same as previous example

};

main()
{
     use Microsoft;

     // These statements are equivalent to those in the
     // previous example.

     Window myWindow;

     windowCount = 1;

     tallyWindow(myWindow);
}
```

Now the '::' operator is only needed in those cases where a symbol is used in two bundles. For example:

```
bundle VendorA {

     class List { /* List Box stuff */ };
};

bundle VendorB {

     class List { /* Seq Ordered collection stuff */ };
};

void organizeMenus ()
{
     VendorB::List mylist;
     VendorA::List * menu;

     // ...
}
```

## 4.0 Summary

Name pollution will become a serious problem for the C++ programmers of tomorrow. Organizing libraries of classes into bundles will keep the name spaces uniquely identifiable and, with the use statement, will not burden the consumer of these libraries with awkward notation.

I would like to thank Martin O'Riordan for his valuable input to this proposal.