Using “using” - How to Use the std Namespace

David Kieras
EECS Department, University of Michigan
Prepared for EECS 381, Fall 2006

Why Namespaces?

When programs get very large and complex, and make heavy use of libraries from a variety of sources, the possibility of *name collisions* rears its incredibly annoying head. A name collision is when an identifier (say a function name) is used in one part of the code to refer to something, but that same identifier already is being used in a different part of the code to refer to something else. Often the programmer can just use a different name for one of them, but if the two conflicting names are defined in two big expensive libraries that are both needed for the project, the programmer is stuck — the compiler won't let the same name be used for two different things!

A good example is a class or struct for representing points in Cartesian space with x, y coordinates. What's the obvious name for this? “Point”, or perhaps “point”. Where might such a thing be used? In the GUI library, to refer to points on the display, normally as a pair of integers. Also in say a geometry/trigonometry library to refer to points on the plane, normally as a pair of double-precision floating point numbers. Oops! Two definitions of “Point” are now in play. If we want to write code using the trig library to determine lines and shapes to put on the display, we are stuck with using both versions of Point at once, so the name collision can't be avoided. If we are lucky, we won't have the problem because one of the library developers used something like CPoint instead of Point, but that is purely a matter of luck. Namespaces can solve the problem in the absence of luck.

The Namespace Concept

The namespace idea originated in other languages (e.g. LISP). The idea is that identifiers can be grouped into separate sets, each set associated with a particular library or body of code, and each such set, or namespace, itself has a name. Thus if the same names appear in two libraries, each of which are in their own namespace, the collision can be resolved by qualifying the names with the namespace name. For example, if we want a GUI point, we might write GUILib::Point, and we might designate a trig library point with TrigLib::Point. Note how this is using the double colon (::), the s*cope qualification operator*, analogously to how it is used for class member names.

To avoid the inconvenience of writing the namespace name all the time, we might want to specify that an unqualified Point means the Point from TrigLib. Then if we want a GUILib Point, we have to use the qualified GUILib::Point name. C++ has “using” statements that provide this convenience.

The *global namespace* is where names reside if you don't put them into a specific namespace. All of the code you normally write will declare functions whose names are in the global namespace. Such names can almost always be used without qualification, but sometimes you have to tell the compiler that a name is in the global namespace to avoid a collision with the same name in another scope or namespace. The name of the global namespace is nothing - literally no name at all – you qualify a name in the global namespace by writing the :: operator with nothing to its left. For example, if “`int foo(int);`” is the prototype of a function in the global namespace, and we need to inform the compiler of that fact, we would write:

```
::foo
```

for the name of the function, as in

```
int result = ::foo(i);
```

It appears at this time in the evolution of C++ programming, that the main reason for creating a namespace is to package a large and complex library to help prevent name collisions and make it easier to tell what is part of the package. However, creating a library is a relatively unusual activity for most programmers, compared to using existing libraries to solve application problems. Thus almost all of the time, your use of namespaces is limited to making use of libraries that declare namespaces. Therefore, this document does not deal with how you declare a namespace or put things into a namespace, only with how to use an existing namespace. The Standard Library namespace, *std*, is the most important existing namespace, but the concepts and guidelines in this document apply to using any predefined library namespace.

Here is a good place to point out that programmers should *never* put something of their own in the std namespace - this namespace is reserved for use only by the official Standard Library. It is a good thing namespaces were invented, because the Standard Library is very large, and its developers did not hesitate to make use of many “good” names for things - so a name collision with the Standard Library can easily happen.

But to use the Standard Library, you have to tell the compiler in some way that you are using names in the std namespace. In fact, in an up-to-Standard compiler and library, you can't even write the "Hello, world!" program without making use of the std namespace explicitly! This document presents the recommended ways to do it.

"Using" Statements

The "full name" of something in the Standard Library has "std::" at the beginning. You can *always* refer to something in the Standard Library with its *fully qualified* name, as in:

```
std::cout << "Hello, World!" << std::endl;
```

Needless to say, this is awkward and verbose, so there are ways to tell the compiler that you are "using" certain names throughout your code file. This is the "using" statement, which comes in two forms:

(1) A *using directive* says you are using an entire namespace, as in:

```
using namespace std;
```

You are directing the compiler to make all of the names in the std namespace part of the global namespace, and thus they can be referred to without qualification in the rest of your source code file. They will now collide with any names that your own code uses. This takes effect only for the current file that is being compiled.

(2) A *using declaration* says you are using only a single name from a namespace, as in:

```
using std::cout;
```

This declares that the "cout" you are referring to is the "std" cout, which is now in the global namespace. Now you can use "cout" without qualification, but no other std names have been made part of the global namespace. This also takes effect only for the current file that is being compiled.

The rules and guidelines presented next are ways to make use of these three forms of namespace reference in the way that minimizes name collision possibilities and maximizes code clarity and writing convenience. The rules differ for header files (.h files) and source (or implementation, .cpp) files.

Guidelines for Header files

#1. Do not put any form of using statement in a header file. The reason? Anybody wanting to use your component has to #include your header file. If you have using statements in it, then these statements become part of their code, appearing at the point your header file was included. They are stuck with whatever namespace using decision you made, and can't override it with their own. Furthermore, the using statement will take effect at the point where it appears in the code that #included the header, meaning that any code processed before that might get treated differently from code processed after that point. There might be a hodgepodge of which headers and code gets interpreted in terms of your namespace decision.

A single "using namespace std;" statement in a single header file in a complex project can make a mess out of the namespace management for the whole project. So, *no "using" in a header file!*

#2. Use fully qualified names for Standard Library names in header files. Of course, if you have classes or functions that refer to the Standard Library classes, you have to be able to name them in the header file. Since you can't put a using statement in the header file, you must use a fully qualified name for Standard Library classes or objects in the header file. Thus, expect to see and write lots of `std::string, std::cout, std::ostream`, etc. in header files.

Guidelines for Source (Implementation) Files

#3. Put all using statements* after *all #includes. This first rule is related to the "no using in headers" rule: Do not put any "using" statements before any #includes of header files; all "usings" should come after all of the #includes. For example, suppose you break this rule as follows:

```
#include <iostream>
using namespace std;       // Warning! Danger! Potential Evil!
#include <GUILib.h>
```

Recall that what #include does is essentially a copy-paste of the entire text of the header file into that place in this file, and then the compiler processes this entire mass of text. By putting the "using" before the GUILib #include, you have a situation in which the GUILib.h header file (and all the headers it includes) will be interpreted in a context where all of the std namespace names are in the global namespace. This has essentially the same potential for evil as putting "using namespace std;" directly in the GUILib.h file itself.

Instead, the above should be:

```
#include <iostream>
#include <GUILib.h>
// all other #include's
using namespace std;       // no harm done to headers
```

The same rule applies for using-declarations such as "using std::cout;" although the potential for mischief is lower.

#4. Say "using namespace std;" for maximum clarity and convenience and no collision protection. You have a choice for how much of std you want to put into the global namespace: all, part, or none. The namespace directive "using namespace std;" puts all of the std namespace into the global namespace, meaning that you can freely refer to everything in the Standard Library without having to do any qualifications.

This blanket using directive by far makes your code easiest to write and generally easiest to read (if the code reader is familiar with the Standard Library). However, namespace collisions are now more likely and you will have to deal with them. But you can't beat the clarity and convenience; you'll find lots of people doing this, especially for code that uses the Standard Library heavily.

#5. Use fully qualified names everywhere only if you like colons poking you in the eye and like to frustrate other people, one of whom is me, who doesn't like it at all, and gets annoyed while grading projects. Some people like to put *none* of std into the global namespace, and so they never use a "using" statement – they like to write fully qualified names throughout their code, including their implementation files. They never have to worry about name collisions because they never use a name that might collide! Such code is full of "std::" as in:

```
std::string foo(std::map<std::string, std::greater<std::string>, std::string>& table)
{
std::string s1, s2;
std::cout << "Enter two strings:" << std::endl;
std::cin >> s1 >> s2;
std::map<std::string, std::greater<std::string>, std::string>::iterator it = table.find(s1);
// etc.
```

When you read such code, the colons tend to stand out and produce a remarkably distracting effect like the page has been machine-gunned. Most people find such code much harder to read. Others apparently enjoy it - the sheer geekiness of it is amazing. It does make it painfully obvious which things are from the Standard Library, and so can help the reader who is less familiar with the Standard Library. But it is also a lot more typing. My recommendation is not to use this approach; it is legal, but it is overkill, and some people (like me) absolutely hate it. Bottom line: *Don't do it in this course*.

#6. Recommended: Use specific using declarations for just the names you need. Instead of a blanket "using namespace std;" directive, write using declarations for only the components of the Standard Library that your code is actually using in this file. This provides complete name collision protection for all of the other names, and with no loss of clarity and convenience in the code itself. The reader who is not completely familiar with the Standard Library can look at the using declarations for clues about which names are coming from the Standard Library.

This is not as much work as it might seem. Notice that using a class automatically results in using all of its members; so to use all of std::string, a single

```
using std::string;
```

will suffice. Unfortunately, using console I/O involves listing all of the objects and manipulators, because they aren't all members of a single class:

```
using std::cin; using std::cout; using std::endl;
```

I like to group the using declarations for related names on a single line.

My experience has been that you don't need very many using declarations, even in code that makes heavy use of the Standard Library.