C++ Without genlib.h

When you arrived at your first serious CS106B or CS106X lecture, you probably learned to write a simple "Hello, World" program like this one shown below:

```
#include "genlib.h"
#include <iostream>

int main()
{
      cout << "Hello, world!" << endl;
      return 0;
}
```

Whether or not you have previous experience with C++, you probably realized that the first line means that the source code references an external file called genlib.h. For the purposes of CS106B/X, this is entirely acceptable (in fact, it's required!), but once you migrate from the educational setting to professional code you might run into trouble because genlib.h is *not* a standard header file. It's part of the CS106 libraries in order to simplify out certain language features so you can focus on writing code, rather than appeasing the compiler.

In CS106L, none of our programs will use genlib.h, simpio.h, or any of the other CS106 library files. Don't worry, though, because none of the functions exported by these files are "magical." In fact, over the course of CS106L you will learn how to rewrite or supersede the functions and classes exported by the CS106 libraries.* If you have the time, I encourage you to actually open up the genlib.h file and peek around at its contents.

To write "Hello, World" without genlib.h, you'll need to add another line to your program. The "pure" C++ version of "Hello, World" thus looks something like this:

```
#include <iostream>
using namespace std;

int main()
{
      cout << "Hello, World!" << endl;
      return 0;
}
```

The only major difference is that the header file genlib.h has been replaced by the somewhat cryptic statement "`using namespace std;`" Before explaining exactly what this statement does, we need to take a quick diversion to lessons learned from development history.

---

* The exceptions are the graphics and sound libraries. C++ does not have natural language support for multimedia, and although many such libraries exist, we won't cover them in this class.

Let's suppose you're working at a company that produces two types of software: graphics design programs and online gunfighter duels (admittedly, this combination is pretty unlikely, but humor me for a while). Each project has its own source code files complete with a set of useful helper functions and classes. Here are some sample header files from each project, with most of the commenting removed:

GraphicsUtility.h:

```
/* File: graphicsutility.h
 * Graphics utility functions.
 */

/* ClearScene: Clears the current scene. */
void ClearScene();

/* AddLine: Adds a line to the current scene. */
void AddLine(int, int, int, int);

/* Draw: Draws the current scene. */
void Draw();
```

GunfighterUtility.h:

```
/* File: gunfighterutility.h
 * Gunfighter utility functions.
 */

/* MarchTenPaces: Marches ten paces, animating each step. */
void MarchTenPaces(PlayerObject &);

/* FaceFoe: Turns to face the opponent. */
void FaceFoe();

/* Draw: Unholsters and aims the pistol. */
void Draw();
```

Now, suppose that the gunfighter team is implementing `MarchTenPaces` and needs to animate the gunfighters walking away from one another. Realizing that the graphics team has already made a whole set of library functions for graphics, the gunfighter programmers import graphicsutility.h into their project, write code using the graphics functions, and try to compile. However, when they try to test their code, the compiler spits out errors, which might look something like "`error: function 'void Draw()' already defined.`"

The problem exists because both the graphics and the gunfighter modules contain functions named `Draw()` with the same parameters and the compiler can't distinguish between them. Unfortunately, it's impractical for either team to rename their `Draw` function, both because the other programming teams expect them to provide functions named `Draw` and because their code is already filled with calls to `Draw`. Fortunately, there's an elegant solution to this whole problem.

Enter the C++ `namespace` keyword. A namespace adds another layer of naming onto your functions and variables. For example, if all of the gunfighter code was in the namespace "Gunfighter," the function `Draw` would have the full name `Gunfighter::Draw`. Similarly, if the graphics programmers put their code inside namespace "Graphics," they would reference the function `Draw` as `Graphics::Draw`. If this is the case, there won't be any ambiguity between the two functions, and the gunfighter development team would be able to compile their code normally.

But there's still one problem – other programming teams are expected to call functions named `ClearScene` and `FaceFoe`, not `Graphics::ClearScene` and `Gunfighter::FaceFoe`. Fortunately, C++ allows what's known as a `using` declaration that lets you ignore fully-qualified names from a namespace and instead use the shorter names.

Back to the Hello, World example (reprinted here)

```
#include <iostream>
using namespace std;

int main()
{
      cout << "Hello, World!" << endl;
      return 0;
}
```

The statement "`using namespace std;`" that follows the `#include` directive tells the compiler that in your program, all of the functions and classes in the namespace "std" can be used without their fully-qualified names. This "std" namespace is the C++ standard namespace that includes all of the library functions and classes of the C++ Standard Library. For example, `cout` is truly named `std::cout`, and without the using declaration importing the std namespace, Hello, World would look something like this:

```
#include <iostream>

int main()
{
      std::cout << "Hello, World!" << std::endl;
      return 0;
}
```

Clearly this can get to be a bit of a hassle, so in genlib.h we included the using declaration for you. But now that we've taken the training wheels off and genlib.h is no more, you'll have to remember to include it yourself!

There's one more part of genlib.h that's important to note, and that's the use of the `string` type. Unlike other programming languages, in C++ there is no primitive string type. Sure, there's the class `string`, but unlike an `int` or a `double`, it's not a built-in type and thus must be included with a `#include` directive. Specifically, you'll need to write `#include <string>` at the top of any program that wants to use C++-style strings. And don't forget the using declaration, or you'll need to write `std::string` every time you want to use C++ strings!