Chapter 2: C++ without `genlib.h`

When you arrived at your first CS106B/X lecture, you probably learned to write a simple "Hello, World" program like the 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 will run into trouble because `genlib.h` is *not* a standard header file; it's included in the CS106B/X libraries to simplify 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 CS106B/X library files. Don't worry, though, because none of the functions exported by these files are "magical." In fact, in the next few chapters you will learn how to rewrite or supersede the functions and classes exported by the CS106B/X 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;
}
```

We've replaced the header file `genlib.h` with the 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. 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 helper functions and classes. Here are some sample header files from each project, with most of the commenting removed:

---

* 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 text.

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 x0, int y0, int x1, int y1);

/* 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 &toMove);

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

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

Suppose the gunfighter team is implementing `MarchTenPaces` and needs to animate the gunfighters walking away from one another. Realizing that the graphics team has already implemented an entire library geared toward this, 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 linker reports errors to the effect of "`error: function 'void Draw()' already defined`."

The problem is that the graphics and gunfighter modules each contain functions named `Draw()` with the same signature and the compiler can't distinguish between them. 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 resolution to this 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 is no longer any ambiguity between the two functions, and the gunfighter development team can compile their code.

But there's still one problem – other programming teams expect to find 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;`" following the `#include` directive tells the compiler that 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 the library functions and classes of the 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;
}
```

While some programmers prefer to use the fully-qualified names when using standard library components, repeatedly writing `std::` can be a hassle. To eliminate this problem, 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 important part of `genlib.h`, the `string` type. Unlike other programming languages, C++ lacks a primitive string type.* Sure, there's the class `string`, but unlike `int` or `double` it's not a built-in type and 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!

* Technically speaking there are primitive strings in C++, but they aren't objects. See the chapter on C strings for more information.