BewareOfStringPromotion

`std::string` is a useful tool for simplifying memory management of strings and avoiding unnecessary copies by reference counting. However there is one common gotcha where it *causes* unnecessary copies. Consider:

```
void f(const std::string& s) { cout << s << endl };

void g() {
  for (int i = 0; i <  1000; ++i) { f("hello"); };
}
```

This actually allocates, copies and deletes 1000 heap buffers with the string "hello"! The problem here is that "hello" is *not* an instance of `std::string`. It is a char[5] that must be converted to a temporary `std::string` using the appropriate constructor. However `std::string` always wants to manage its own memory, so the constructor allocates a new buffer and copies the string. Once f() returns and we go round the loop again the temporary is deleted along with its buffer.

Here's a better solution:

```
void f(const std::string& s) { cout << s << endl };
namespace { const std::string hello("hello"); }
void g() {
  for (int i = 0; i <  1000; ++i) { f(hello); };
}
```

This time we have a constant `std::string` that is created once at start up and destroyed once at shut-down. The anonymous namespace makes the constant private to this .cpp file so we wont have name clashes. (Its similar to using the static keyword on a global declaration in C, but anonymous namespaces are the preferred way to do it in modern C++)