Thread-Safe Function-Scope `static` Variables

Initialization of function-scope **`static`** objects is now guaranteed to be free of data races in the presence of multiple concurrent threads.

Description

A variable declared at function, a.k.a. local, scope has **automatic storage duration**, except when it is marked **`static`**, in which case it has **static storage duration**. Variables having automatic storage duration are allocated on the stack each time the function is invoked and initialized when that invocation's **flow of control** passes through the **definition** of that object. In contrast, variables with static storage duration, e.g., `iLocal`, defined at function scope, e.g., `f`, are instead allocated once per program and are initialized only the first time the flow of control passes through the definition of that object:

```
#include <cassert>  // standard C assert macro

int f(int i) // function returning the first argument with which it is called
{
    static int iLocal = i;  // Object is initialized only once, on the first call.
    return iLocal;          // The same iLocal value is returned on every call.
}

int main()
{
    int a = f(10);  assert(a == 10);  // Initialize and return iLocal.
    int b = f(20);  assert(b == 10);  // Return iLocal.
    int c = f(30);  assert(c == 10);  // Return iLocal.

    return 0;
}
```

In the simple example above, the function, `f`, initializes its **`static`** object, `iLocal`, with its argument, `i`, only the first time it is called and then always returns the same value, e.g., 10. Hence, when that function is called repeatedly with distinct arguments to initialize the `a`, `b`, and `c` variables, all three of them are initialized to the same value, 10, supplied to the first invocation of `f`. Although the function-scope **`static`** object, `iLocal`, was created after `main` was entered, it will not be destroyed until after `main` exits.

Concurrent initialization

Historically, initialization of **function-scope** static storage duration objects was not guaranteed to be safe in a **multithreading context** because it was subject to **data races**