Lecture 12: Transitioning from C to C++ Threading

- Introverts Revisited, in C++
  - Rather than deal with pthreads as a platform-specific extension of C, I'd rather use a thread package that's officially integrated into the language itself.
    - As of 2011, C++ provides support for threading and many synchronization directives.
    - Because C++ provides better alternatives for generic programming than C does, we avoid the void * tomfoolery required when using pthreads.
  - Presented below is the object-oriented C++ equivalent of the introverts example we've already seen once before. The full program is online right here.

```cpp
static void recharge() {
    cout << oslock << "I recharge by spending time alone." << endl << osunlock;
}

static const size_t kNumIntroverts = 6;
int main(int argc, char *argv[]) {
    cout << "Let's hear from " << kNumIntroverts << " introverts." << endl
    thread introverts[kNumIntroverts]; // declare array of empty thread handles
    for (thread& introvert: introverts) // move anonymous threads into empty handles
        introvert = thread(recharge);
    cout << "Everyone's recharged!" << endl;
    return 0;
}
```
Lecture 12: Transitioning from C to C++ Threading

- We declare an array of empty `thread` handles as we did in the equivalent C version.
- We install the `recharge` function into temporary threads that are then moved (via the thread's `operator=(thread&& other)`) into a previously empty `thread` handle.
  - This is a relatively new form of `operator=` that fully transplants the contents of the `thread` on the right into the `thread` on the left, leaving the `thread` on the right fully gutted, as if it were zero-arg constructed. Restated, the left and right thread objects are effectively swapped.
  - This is an important distinction, because a traditional `operator=` would produce a second working copy of the same `thread`, and we don't want that.
- The `join` method is equivalent to the `pthread_join` function we've already discussed.
- The prototype of the thread routine—in this case, `recharge`—can be anything (although the return type is always ignored, so it should generally be `void`).
- `operator<<`, unlike `printf`, isn't thread-safe.
  - I've constructed custom stream manipulators called `oslock` and `osunlock` that can be used to acquire and release exclusive access to an `ostream`.
  - These manipulators—which we can use by `#include-ing "ostreamlock.h"`—can be used to ensure at most one thread has permission to write into a stream at any one time.
Lecture 12: Transitioning from C to C++ Threading

- Thread routines can accept any number of arguments using variable argument lists. (Variable argument lists—the C++ equivalent of the ellipsis in C—are supported via a recently added feature called **variadic templates**.)
- Here's a **slightly more involved example**, where `greet` threads are configured to say hello a variable number of times.

```cpp
static void greet(size_t id) {
    for (size_t i = 0; i < id; i++) {
        cout << oslock << "Greeter #" << id << " says 'Hello!'" << endl << osunlock;
        struct timespec ts = {
            0, random() % 1000000000
        };
        nanosleep(&ts, NULL);
    }
    cout << oslock << "Greeter #" << id << " has issued all of his hellos, " << "so he goes home!" << endl << osunlock;
}

static const size_t kNumGreeters = 6;
int main(int argc, char *argv[]) {
    cout << "Welcome to Greetland!" << endl;
    thread greeters[kNumGreeters];
    for (size_t i = 0; i < kNumGreeters; i++) greeters[i] = thread(greet, i + 1);
    for (thread& greeter: greeters) greeter.join();
    cout << "Everyone's all greeted out!" << endl;
    return 0;
}
```