Simple Concurrency In C++

Paul Bauer

Colorado State University Global

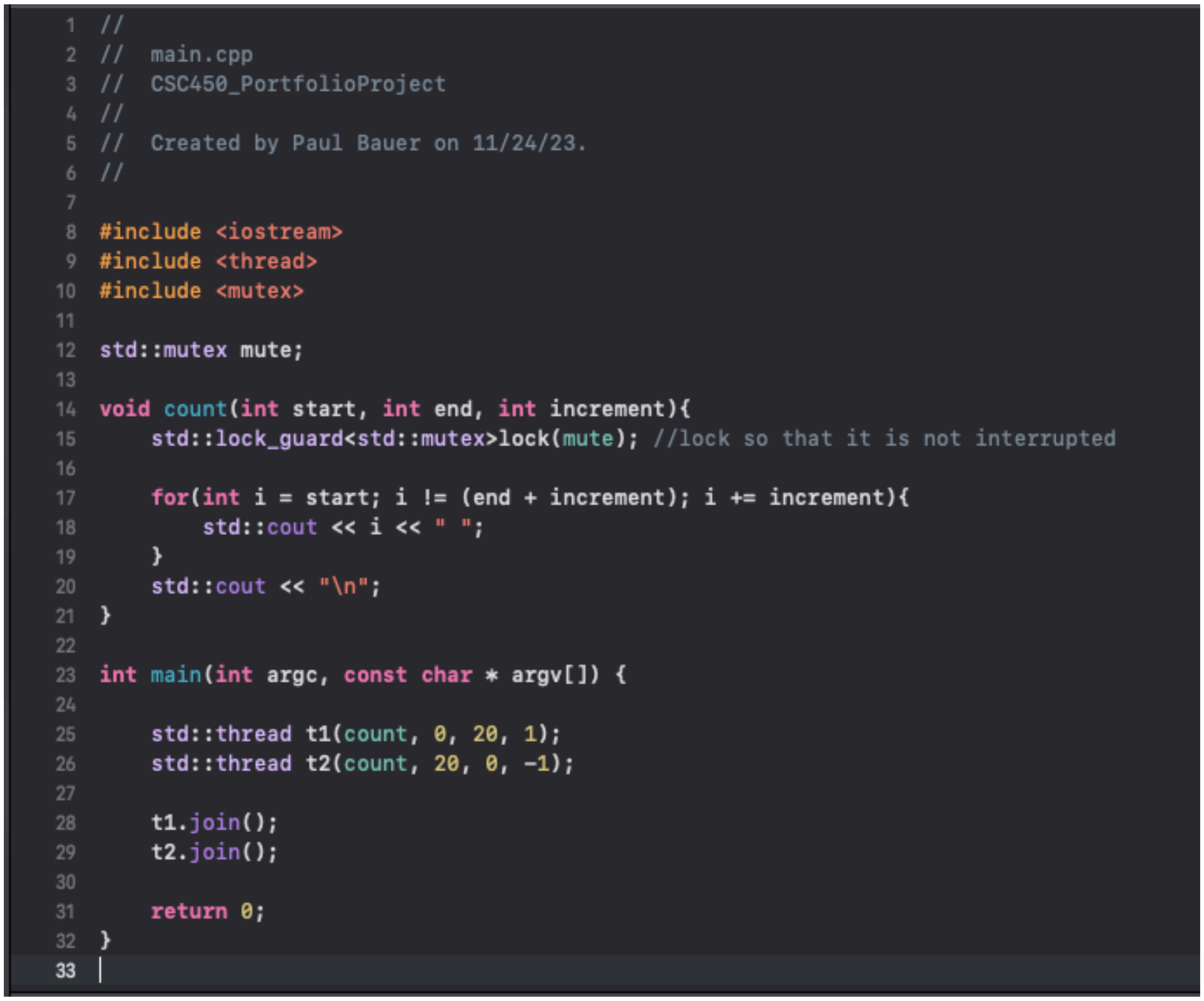
CSC450: Programming III

Reginald Haseltine

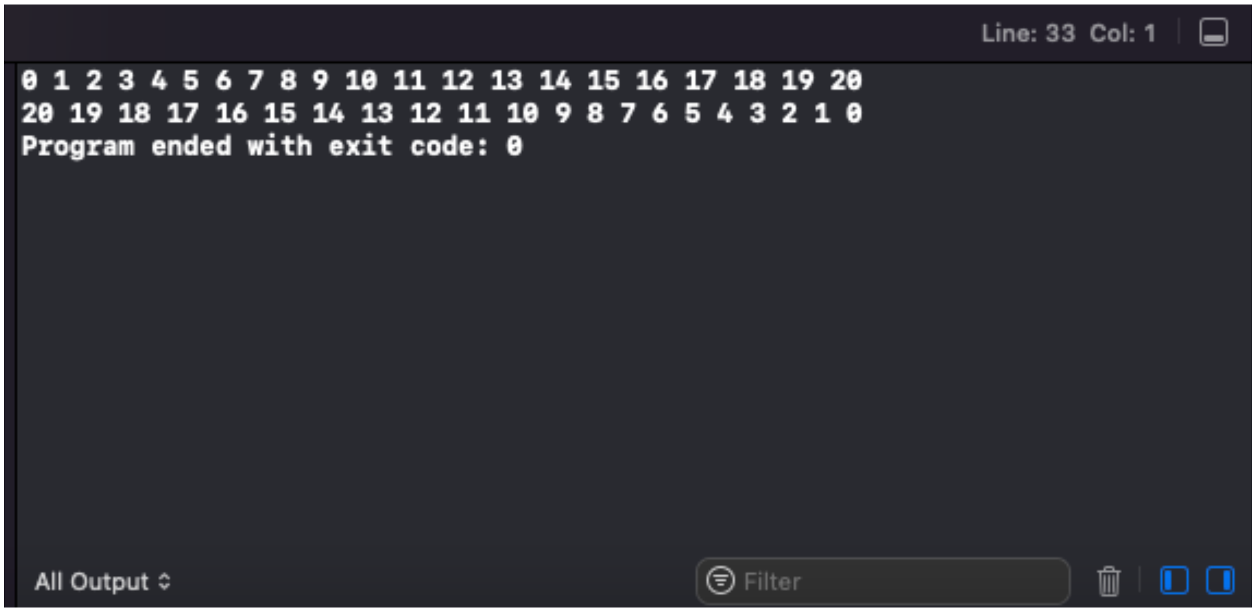
November 26th, 2023

Simple Concurrency In C++

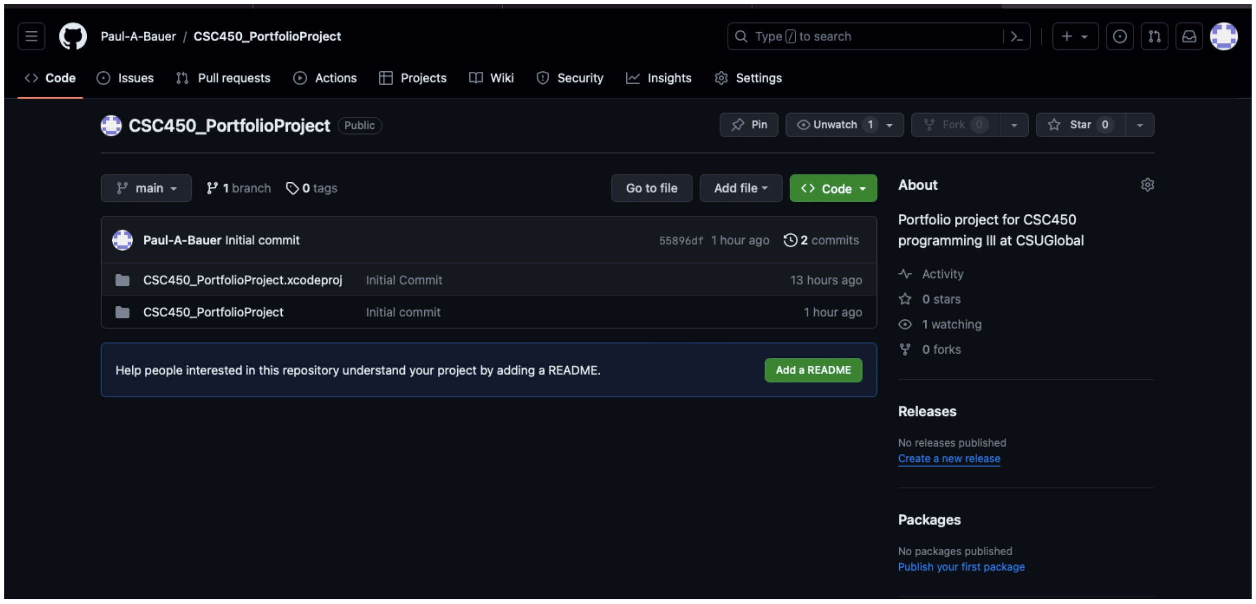
This C++ program demonstrates a basic example of concurrency using std::thread. The program starts two threads that each execute the count function with different input parameters, one counting up from zero to twenty and one counting down from twenty to zero. Because the count function accesses std::cout it is important to make sure that it has exclusive access to avoid race conditions. This is done using an std::mutex variable which is defined globally to insure the destructor is not called while it is still in use by another thread. Locking the mutex before accessing std::cout using std::lock_guard prevents other threads attempting to lock the same mutex from proceeding until this one has released the mutex. The first thread locks the mutex while counting up to twenty and then then releases it by unlocking the mutex automatically. After the mutex is unlocked the second thread can lock it and count down from twenty to zero. For more complex functions it would be preferred to explicitly unlock the mutex for several reasons, most notably because if there is a possibility of exceptions being thrown it is important to insure the mutex is always unlocked. Another good reason to unlock the mutex manually would be because it is important to release the mutex as soon as the function is finished using the shared resources. If there were multiple mutex variables being used to monitor access to different resources it would be essential to coordinate the order they are locked and released to prevent deadlock as well.



```cpp
//
//  main.cpp
//  CSC450_PortfolioProject
//
//  Created by Paul Bauer on 11/24/23.
//

#include <iostream>
#include <thread>
#include <mutex>

std::mutex mute;

void count(int start, int end, int increment){
    std::lock_guard<std::mutex>lock(mute); //lock so that it is not interrupted

    for(int i = start; i != (end + increment); i += increment){
        std::cout << i << " ";
    }
    std::cout << "\n";
}

int main(int argc, const char * argv[]) {

    std::thread t1(count, 0, 20, 1);
    std::thread t2(count, 20, 0, -1);

    t1.join();
    t2.join();

    return 0;
}

```



```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Program ended with exit code: 0
```



CSC450_PortfolioProject Public

Paul-A-Bauer Initial commit	55896df 1 hour ago	2 commits
CSC450_PortfolioProject.xcodeproj	Initial Commit	13 hours ago
CSC450_PortfolioProject	Initial commit	1 hour ago

Help people interested in this repository understand your project by adding a README.

About

Portfolio project for CSC450 programming III at CSUGlobal

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

https://github.com/Paul-A-Bauer/CSC450_PortfolioProject

Conclusion

This example is very simple but demonstrates some of the key concepts of concurrency and how to handle the common vulnerabilities associated with concurrency. Using mutex variables to manage access to resources and coordinating the order they are locked and unlocked using std::lock_gaurd insures that shared resources will be accessed by only one thread at a time and in an appropriate order. For more typical threading applications that will complete more complex operations it is also essential to lock mutex variables only for the portion of a function that accesses shared resources and unlock in all possible control flow paths including error handlers that will terminate the program or throw a new error.

References