Security and Concurrency in C/C++

Stephan Peters

Colorado State University Global

CSC450-1 23WD: Programming III

Reginald Haseltine

31 March 2024

Security and Concurrency in C/C++

This paper will briefly cover security concepts introduced throughout the course and will demonstrate a C++ program that utilizes threads sequentially rather than concurrently. Two topics that will be covered include performance issues with concurrency and vulnerabilities exhibited with use of strings. After this, I will demonstrate a C++ program which uses threads to count up and count down in a sequential manner, then I will discuss the security of the data types exhibited in the program.

Performance Issues with Concurrency

Concurrency refers to executing several computations simultaneously. Running multiple simultaneous threads is a simple task if they do not interact with mutable shared objects. Concurrency can allow your application to accomplish more work in less time, though this is not always the case. Outside of the normal issues that may be caused from incorrectly programming for concurrency, such as unsynchronized collections, race conditions, memory inconsistencies (especially stemming from use of memory caches by the OS), non-atomic variables, deadlock, etc. (see Burcea, 2019) using functions such as mutex may actually decrease the performance of your application.

In an experiment performed by Sidath Munasinghe using C++, he found using concurrent threads to speed up small computation problems will actually decrease the performance of the application due to additional overhead caused by:

1. Time to initialize
2. Time to finalize
3. Overhead due to external libraries

4. Overhead caused by communication between threads

(Munasinghe, 2018)

Figure 1 shows the result of running code shown in the reference with 5 insert operations, 5 delete operations, and 990 member operations.

Figure 1

Experiment Results (Munasinghe, 2018)

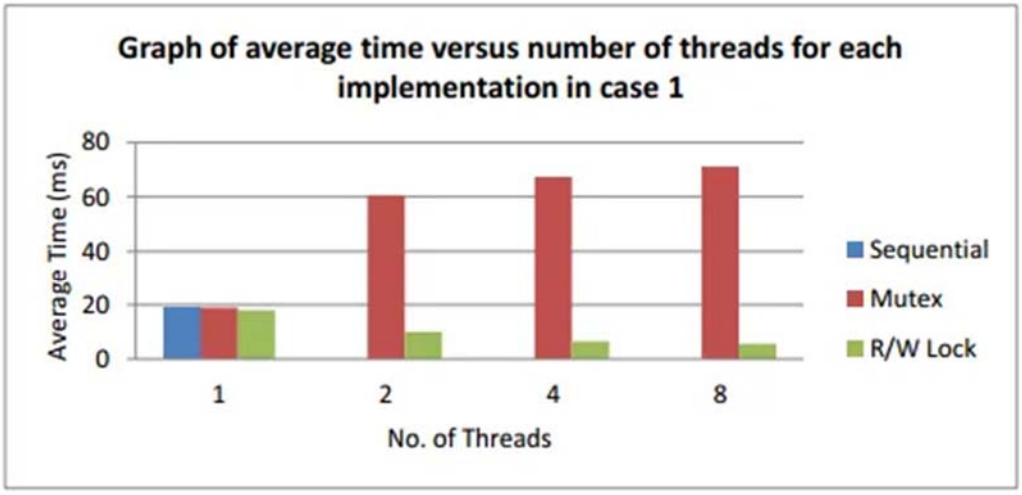


Figure 1- Case1

Vulnerabilities in Strings

The most egregious string vulnerabilities are with "C-style strings," which may be used in C++ programs. Common string manipulation errors include:

- Unbounded string copies
- Null-termination errors
- Truncation
- Write outside array bounds
- Off-by-one errors
- Improper data sanitization

(Seacord, 2013 § 2.2)

Unbounded string copies occur when copying from a source to a fixed length character array which leads to undefined behavior a malicious attacker can use. Null-termination errors occur if a string is not properly terminated at the last element of an array. A malicious attacker can use this to read or write data outside the bounds of the array. Truncation occurs if a container array is not large enough to carry the contents of a string copied to it. This can lead to data loss. Writing outside array bounds, as in unbounded string copies can allow a malicious attacker to write directly to memory, including executable commands. Off-by-one errors are another way an attacker can write data outside of array bounds, even if the array is properly terminated. It is in essence like an unbounded string copy. Improper data sanitization occurs if data comes from an untrusted source and has not been properly sanitized, so it conforms to any constraints on the target value.

A C++ Counting Program that Uses Threads Sequentially

I found in C++, while a thread can notify other listening threads it is complete, it is not possible to determine which thread in a pool of threads starts first, and the threads must be forced to start sequentially to determine start order in a thread pool. As the threads are forced to be sequential by the instructions of the assignment, I left mutex locks unused, though they are present in the program.

My initial approach was a non-static Count class where the object stores a minimum, maximum and increment value, and it includes methods to count up and count down. Within those methods is an optional mutex lock and thread notifier. In order to run non-static methods with a thread, a worker class must be constructed that allows object references to be used within

the thread signature. Source Code 1 shows the contents of Count.h, which houses the object, the constructor, and the methods used to count up and down.

Source Code 1

The Count class: Count.h.

```
#include <iostream>
#include <mutex>
#include <condition_variable>

using namespace std;

/*
 * An object class that stores a minimum, a maximum,
 * and an incremental value for counting.
 * Class methods allow a countUp or countDown
 * with or without a mutex lock to wait for other threads to complete.
 */
class Count {
public:
    // Constructs the object and initializes the variables
    explicit Count(int min = 1, int max = 20, int increment = 1)
            : min_(min), max_(max), increment_(abs(increment)) {}

    // Initiates countUp() with wait false as a default.
    void countUp() {
        countUp(false);
    }
    // Counts from the minimum to the maximum.
    // Initiates a thread-lock if wait is true.
    // Notifies listening threads the thread is complete.
    void countUp(bool wait){
        if (wait) {
            unique_lock<mutex> lock(mutex_);
            cv_.wait(lock, [this] { return count_ == max_; });
        }
        for (int i = min_; i <= max_; i += increment_) {
            cout << i << " ";
        }
        cout << endl;
        count_ = min_;
        cv_.notify_all();
    }

    // Initiates countDown() with wait false as a default.
    void countDown() {
        countDown(false);
    }
    // Counts from the maximum to the minimum.
    // Initiates a thread-lock if wait is true.
    // Notifies listening threads the thread is complete.
    void countDown(bool wait) {
        if (wait) {
```

```
            unique_lock<mutex> lock(mutex_);
            cv_.wait(lock, [this] { return count_ == min_; });
        }
        for (int i = max_; i >= min_; i -= increment_) {
            cout << i << " ";
        }
        cout << endl;
        count_ = max_;
        cv_.notify_all();
    }

private:
    // Initialize variables that can be used as references.
    int min_, max_, increment_;
    int count_ = min_;
    mutex mutex_;
    condition_variable cv_;
};
```

The above class allows the object to be created and allows the parameters stored in the object to be used to count up or down.

The driver class (Count.cpp) includes the worker function that allows the threads to use non-static functions. It is shown in Source Code 2.

Source Code 2

The driver program: Count.cpp.

```
/*
 * Creates two threads that will act as counters.
 * One thread counts up to 20.
 * Once the first reaches 20, then a second thread counts down to 0.
 * Demonstrates a non-static object method with a worker function,
 * and a static method using static functions.
 * (Count.cpp)
 */
#include "Count.h"
#include "StaticCount.h"

using namespace std;

// This is a worker function that allows the threads to use non-static
functions.
auto worker(int min, int max, int increment, bool descending, bool wait) {
    // Initialize variables.
    int min_ = min, max_ = max;
    int count_;
    mutex mutex_;
    condition_variable cv_;
```

```
    // Create a non-static Count object.
    Count counter(min, max, increment);

    // Counts from the maximum to the minimum.
    // Initiates a thread-lock if wait is true.
    // Notifies listening threads the thread is complete.
    if (descending) {
        if (wait) {
            unique_lock<mutex> lock(mutex_);
            cv_.wait(lock, [&count_, &min_] { return count_ == min_; });
        }
        counter.countDown();
        count_ = min_;
        cv_.notify_all();
    // Counts from the minimum to the maximum.
    // Initiates a thread-lock if wait is true.
    // Notifies listening threads the thread is complete.
    } else {  // if(!descending)
        if (wait) {
            unique_lock<mutex> lock(mutex_);
            cv_.wait(lock, [&count_, &max_] { return count_ == max_; });
        }
        counter.countUp();
        count_ = max_;
        cv_.notify_all();
    }
}

// Main driver class that shows the count up and count down
// using non-static and static methods.
int main() {
    // Initialize the minimum, maximum, and increment.
    int min = 1, max = 20, increment = 1;

    // Using a non-static object through a worker function.
    cout << "This demonstrates using a non-static class object for the
threads." << endl;
    thread t1(worker, min, max, increment, false, false);
    t1.join();
    thread t2(worker, min, max, increment, true, false);
    t2.join();

    // Using static functions.
    cout << endl << "This demonstrates using a static functions for the
threads." << endl;
    thread t3(staticCountUp, min, max, increment);
    t3.join();
    thread t4(staticCountDown, min, max, increment);
    t4.join();
    return 0;
}
```

I also chose to include static examples, invoked by the threads t3 and t4 in the above code (Source Code 2). The static method is included in StaticCount.h and may be seen in Source Code 3.

Source Code 3

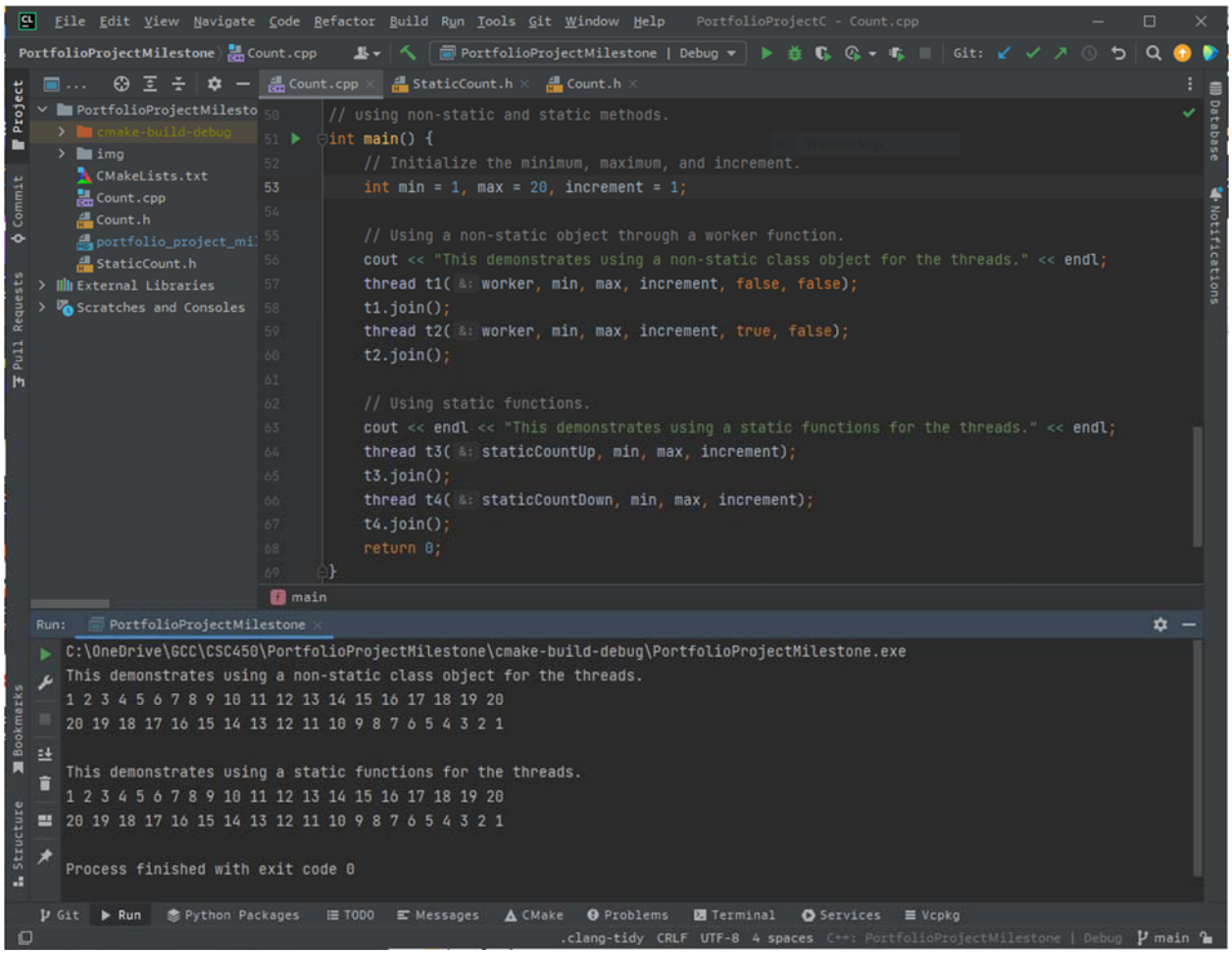
Static method example: StaticCount.h.

```
#include <iostream>
#include <mutex>

using namespace std;

mutex mtx;
static void staticCountUp(int min, int max, int increment){
    for(int i = min; i <= max; i += abs(increment)){
        lock_guard<std::mutex> lock(mtx);
        cout << i << " ";
    }
    cout << endl;
}

static void staticCountDown(int min, int max, int increment) {
    for(int i = max; i >= min; i -= abs(increment)){
        lock_guard<std::mutex> lock(mtx);
        cout << i << " ";
    }
    cout << endl;
}
```

Figure 2 shows the program running. It displays the count programs using both the non-static and static methods being used to initiate the counts. As described, the threads are not being used in a typical manner for threads, but rather they are acting independently and sequentially. The code portion displayed in the screenshot shows the main method in Count.cpp.

Figure 2

Running Count.cpp



Vulnerability of Data Types in the Program

If Count (the object class) is not included as a data type, there are three data types used in the program: int, mutex, and a condition variable. One possible vulnerability (or it could be called a bug) that could have caused the functions to go into an infinite loop is if the increment value given as an argument were negative. This was found in testing and forcing the increment value to be abs(increment) in the class constructor and in the for loops of the static functions solved this issue. Integer overflows can be caused by a user entering a value that cannot be

contained within the minimum or maximum value of an integer. While this would be possible to feed such a value to the class, the overflow would merely cause the for loops not to execute due to an overrun. This was verified in testing. A mutex may contain vulnerabilities if it may be destroyed before thread completion (See Carnegie Mellon University Software Engineering Institute CON50-CPP). There are no instances of mutex within local methods, so this is not a concern in this program. The condition variable does present a concern, as it is possible for a condition variable in the Count object or the worker function to "wake up" if shared with different threads(See Carnegie Mellon University Software Engineering Institute CON55-CPP). It is, in fact, shared by two threads (t1 and t2) but it is unused in this program. This code is non-compliant, and further work would need to be done to it to make it compliant.

Conclusion

Concurrency using threads can significantly decrease processing time if programmed correctly, but care needs to be taken when using threads to process smaller pieces of data, as the processing time may actually increase.

Due to vulnerabilities, "C-Style" strings should be considered as deliberately depreciated, should never be used in new programming, and todos with deadlines should be added to existing software using C-Style strings, and modified to be compliant with current security standards before executing new builds.

While the assigned program does not really demonstrate concurrency in any way, as the threads are run individually and sequentially, it does demonstrate some concepts used in concurrent programming. Before going into production, the program would need to overcome the non-compliant conditional variable issue, possibly through adding the conditional variable to the constructor, or using a different method for locking the threads altogether.

GitHub Repository

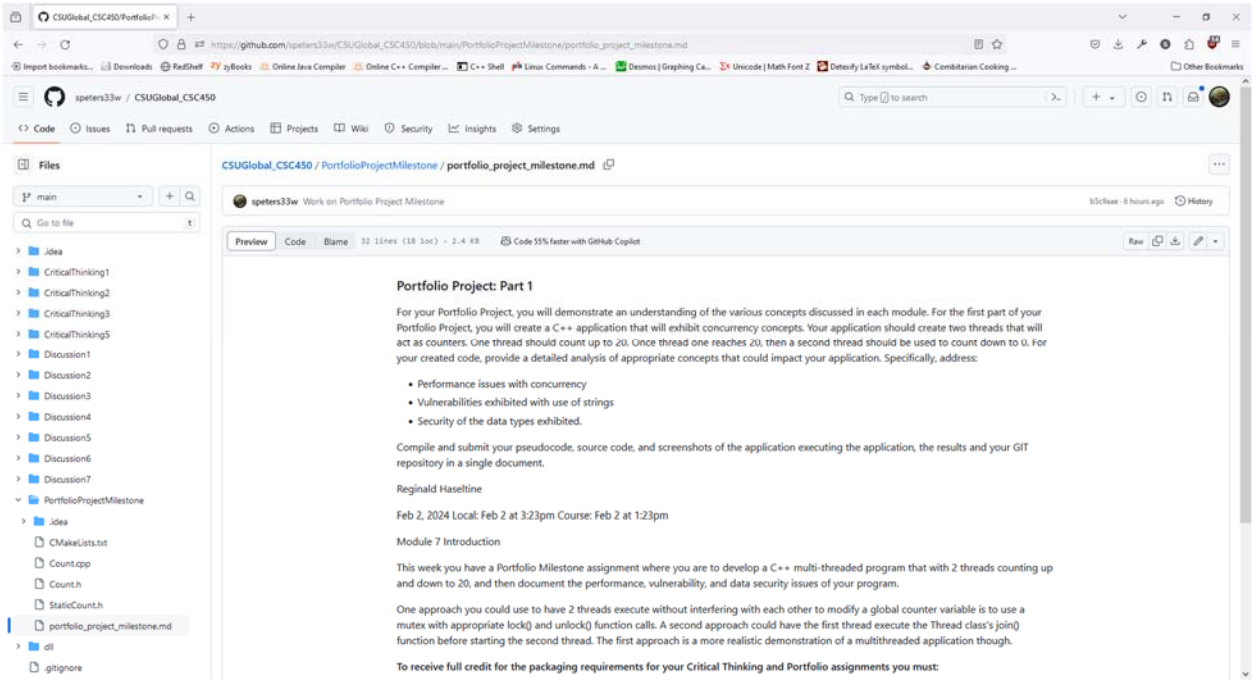
In addition to the solutions above, I was tasked to create a GitHub repository for the project. This repository is located at

https://github.com/speters33w/CSUGlobal_CSC450/tree/main/PortfolioProjectMilestone

Figure 3 shows a screenshot of the main page of this repository.

Figure 3

Image of the main page of my CSC 450 Portfolio Project Milestone GitHub repository



References

Burcea, C. (2019, November 23). *Common Concurrency Pitfalls in Java* (A. Frieze, Ed.). Baeldung. https://www.baeldung.com/java-common-concurrency-pitfalls

Carnegie Mellon University Software Engineering Institute. (2023a, April 20). *CON50-CPP. Do not destroy a mutex while it is locked - SEI CERT C++ Coding Standard - Confluence*. Wiki.sei.cmu.edu. https://wiki.sei.cmu.edu/confluence/display/cplusplus/CON50-CPP.+Do+not+destroy+a+mutex+while+it+is+locked

Carnegie Mellon University Software Engineering Institute. (2023b, December 21). *CON55-CPP. Preserve thread safety and liveness when using condition variables - SEI CERT C++ Coding Standard - Confluence*. Wiki.sei.cmu.edu. https://wiki.sei.cmu.edu/confluence/display/cplusplus/CON55-CPP.+Preserve+thread+safety+and+liveness+when+using+condition+variables

Munasinghe, S. (2018, March 12). *Does Concurrency Really Increase Performance?* Medium. https://towardsdatascience.com/is-concurrency-really-increases-the-performance-8cd06dd762f6

Seacord, R. C. (2013). *Secure coding in C and C++*. Editora: Upper Saddle River Addison-Wesley. https://www.informit.com/articles/article.aspx?p=2036582&seqNum=2