lecture 16

- *C++ thread -continued*
  - *closer look at mutex*
  - *detached vs joinable threads*
  - *lambda functions*
  - *(next lecture) coordination between threads with condition variables*

Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

- C++ threads and counting

```
int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <num_threads> <count_limit>" << std::endl;
        return 1;
    }

    int numThreads = std::stoi(argv[1]); // Number of threads
    const int count_limit = std::stoi(argv[2]); // Number of threads
    std::vector<std::thread> threads; // Vector to store thread objects

    // Spawn threads
    for (int i = 0; i < numThreads; ++i) {
        threads.emplace_back(generateAndCount, i, count_limit); // Each thread has a distinct seed (i)
    }

    // Wait for all threads to finish execution
    for (auto& thread : threads) {
        thread.join();
    }

    std::cout << "Total number of random numbers required to find " << count_limit << " values in the range [0.25, 0.3): " << totalRNGs << std::endl;

    return 0;
}
```

- C++ threads and counting
  - version 1 - is it ok?

```
#include <iostream>
#include <vector>
#include <random>
#include <thread>
#include <mutex>

std::mutex mtx; // Mutex for synchronization
int occurrences = 0; // Global variable to store the total occurrences
int totalRNGs = 0; // Global variable to store the total number of generated random numbers

// Function to generate random numbers and count occurrences
void generateAndCount(int seed, const int count_limit) {
    std::mt19937 rng(seed); // Mersenne Twister random number engine
    std::uniform_real_distribution<double> dist(0.0, 1.0); // Uniform distribution between [0, 1)

    int localCount = 0; // Local variable to store the count of generated random numbers
    while (true) {
        double randomNumber = dist(rng); // Generate a random number
        localCount++; // Increment local count of generated random numbers

        if (occurrences >= count_limit) // Break the loop if the global count of values in the desired range is reached
            break;

        if (randomNumber >= 0.25 && randomNumber < 0.3) {
            mtx.lock(); // Lock the mutex before updating shared variable
            occurrences++; // Increment occurrences if the random number falls within the range [0.25, 0.3)
            mtx.unlock(); // Unlock the mutex after updating shared variable
        }
    }

    mtx.lock(); // Lock the mutex before printing
    std::cout << "Thread " << seed << " generated " << localCount << " random numbers." << std::endl;
    mtx.unlock(); // Unlock the mutex after printing

    mtx.lock(); // Lock the mutex before updating shared variable
    totalRNGs += localCount; // Add local count to global count of generated random numbers
    mtx.unlock(); // Unlock the mutex after updating shared variable
}
```

what about version 2 - is it ok?

```
std::mutex mtx; // Mutex for synchronization
int occurrences = 0; // Global variable to store the total occurrences
int totalRNGs = 0; // Global variable to store the total number of generated random numbers

void generateAndCount(int seed, const int count_limit) {
    std::mt19937 rng(seed); // Mersenne Twister random number engine
    std::uniform_real_distribution<double> dist(0.0, 1.0); // Uniform distribution between [0, 1)

    int localCount = 0; // Local variable to store the count of generated random numbers
    while (true) {
        double randomNumber = dist(rng); // Generate a random number
        localCount++; // Increment local count of generated random numbers

        mtx.lock(); // Lock the mutex before accessing shared variable occurrences
        if (occurrences >= count_limit) { // Check the condition while holding the lock
            mtx.unlock(); // Unlock the mutex if the condition is true
            break;
        }
        if (randomNumber >= 0.25 && randomNumber < 0.3) {
            occurrences++; // Increment occurrences if the random number falls within the range [0.25, 0.3)
        }
        mtx.unlock(); // Unlock the mutex after updating occurrences
    }

    mtx.lock(); // Lock the mutex before printing
    std::cout << "Thread " << seed << " generated " << localCount << " random numbers." << std::endl;
    mtx.unlock(); // Unlock the mutex after printing

    mtx.lock(); // Lock the mutex before updating shared variable totalRNGs
    totalRNGs += localCount; // Add local count to global count of generated random numbers
    mtx.unlock(); // Unlock the mutex after updating totalRNGs
}
```

critical code sections

what could go wrong?

```
if (occurrences >= count_limit) // Break the loop if the global count of values in the desired range is reached
    break;

if (randomNumber >= 0.25 && randomNumber < 0.3) {
    mtx.lock(); // Lock the mutex before updating shared variable
    occurrences++; // Increment occurrences if the random number falls within the range [0.25, 0.3)
    mtx.unlock(); // Unlock the mutex after updating shared variable
}
```

- v1

- v2

```
mtx.lock(); // Lock the mutex before accessing shared variable occurrences
if (occurrences >= count_limit) { // Check the condition while holding the lock
    mtx.unlock(); // Unlock the mutex if the condition is true
    break;
}
if (randomNumber >= 0.25 && randomNumber < 0.3) {
    occurrences++; // Increment occurrences if the random number falls within the range [0.25, 0.3)
}
mtx.unlock(); // Unlock the mutex after updating occurrences
```

```
bash-3.2$ g++ -std=c++17 count-rand-inrange-correct.cpp -o xcount-inrange
bash-3.2$ time ./xcount-inrange
Usage: ./xcount-inrange <num_threads> <count_limit>

real    0m0.317s
user    0m0.002s
sys     0m0.004s
bash-3.2$ time ./xcount-inrange 4 10000000
Thread 2 generated 49430907 random numbers.
Thread 3 generated 49412503 random numbers.
Thread 1 generated 49681921 random numbers.
Thread 0 generated 51391996 random numbers.
Total number of random numbers required to find 10000000 values in the range [0.25, 0.3): 199917327

real    0m16.869s
user    0m20.596s
sys     0m30.200s
bash-3.2$
```

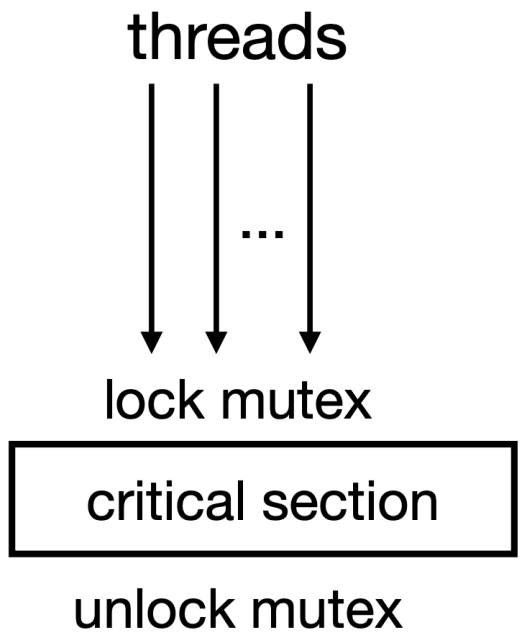
critical sections: mutex passed as reference

```
void call_function_r(const int nt)
{ //nt, number of threads

    std::mutex the_mutex;

    for (int i = 0; i < nt; ++i)
    {
        threads[i] = std::thread(function_r, std::ref(the_mutex), /* args */);
    }

    for (int i = 0; i < nt; ++i)
    {
        threads[i].join();
    }
}
```




```
void function_r(std::mutex &the_mutex, /* other args */)
{
    // do local stuff

    // manage the critical section
    the_mutex.lock();
    // do critical section work here
    // ...
    the_mutex.unlock();
}
```

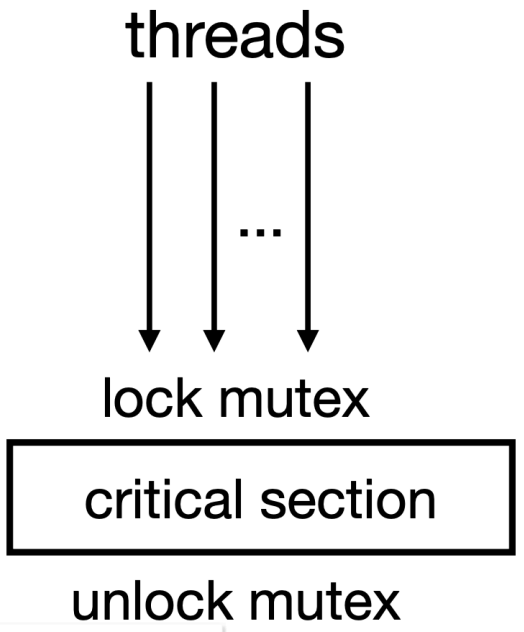
critical sections: lock global variable

```cpp
std::mutex the_mutex; // global

void call_function_r(const int nt)
{ //nt, number of threads

    for (int i = 0; i < nt; ++i)
    {
        threads[i] = std::thread(function_r, /* args */);
    }

    for (int i = 0; i < nt; ++i)
    {
        threads[i].join();
    }
}
```




```cpp
void function_r(/* args */)
{
    // do local stuff

    the_mutex.lock();
    // do critical section work here
    the_mutex.unlock();
}
```

- detached vs joinable threads

A **detached thread** runs independently of the main thread.

Once detached, it **cannot be joined**.

It **continues execution** even if the main thread finishes (unless the process exits).

```
#include <iostream>
#include <thread>

// First void function to be executed by the first thread
void function1() {
    std::cout << "Hello from Function 1" << std::endl;
}

// Second void function to be executed by the second thread
void function2() {
    std::cout << "Hello from Function 2" << std::endl;
}

int main() {
    // Spawn the first detached thread, calling function1
    std::thread([] { function1(); }).detach();

    // Spawn the second detached thread, calling function2
    std::thread([] { function2(); }).detach();

    // Main thread continues execution
    // No need to join detached threads

    // Wait for a moment to allow detached threads to execute
    //std::this_thread::sleep_for(std::chrono::seconds(1));

    return 0;
}
```

- detached threads -safe and unsafe usage

```cpp
#include <iostream>
#include <thread>

void unsafeUse(int& ref) {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Ref: " << ref << std::endl;
}

int main() {
    int x = 42;
    std::thread t(unsafeUse, std::ref(x));
    t.detach();  // Dangerous if x goes out of scope before use

    std::cout << "Main thread exiting early\n";
    return 0;  // x is destroyed here → UB in detached thread!
}
```

`x` goes out of scope before `unsafeUse` reads it.

Causes **undefined behavior** — may crash or print garbage.

```cpp
#include <iostream>
#include <thread>
#include <memory>
#include <chrono>

void safeDetachedTask(std::shared_ptr<int> data) {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Shared value: " << *data << std::endl;
}

int main() {
    auto sharedData = std::make_shared<int>(42);

    std::thread t(safeDetachedTask, sharedData);
    t.detach();  // Safe: sharedData stays alive via ref count

    std::cout << "Main thread done.\n";
    std::this_thread::sleep_for(std::chrono::seconds(2));  // E
    return 0;
}
```

as long as the detached thread actually starts executing and copies the `shared_ptr` before `main()` ends

Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025

```cpp
#include <iostream>
#include <fstream>
#include <thread>

// First void function to be executed by the first thread
void function1() {
    // Create and open file_one.txt
    std::ofstream file("file_one.txt");
    if (file.is_open()) {
        file << "Hello from Function 1" << std::endl;
        file.close();
        std::cout << "File 'file_one.txt' created by Function 1" << std::endl;
    } else {
        std::cerr << "Error: Unable to create file_one.txt" << std::endl;
    }
}

// Second void function to be executed by the second thread
void function2() {
    // Create and open file_two.txt
    std::ofstream file("file_two.txt");
    if (file.is_open()) {
        file << "Hello from Function 2" << std::endl;
        file.close();
        std::cout << "File 'file_two.txt' created by Function 2" << std::endl;
    } else {
        std::cerr << "Error: Unable to create file_two.txt" << std::endl;
    }
}

int main() {
    // Spawn the first detached thread, calling function1
    std::thread([] { function1(); }).detach();

    // Spawn the second detached thread, calling function2
    std::thread([] { function2(); }).detach();

    // Introduce a delay to allow detached threads to execute
    std::this_thread::sleep_for(std::chrono::seconds(1));

    return 0;
}
```

- detached threads -do independent tasks

```
bash-3.2$ g++ -std=c++17 -o xdetach-fil cpp-thread-detach-file.cpp
bash-3.2$ ./xdetach-fil
File 'file_one.txt' created by Function 1
File 'file_two.txt' created by Function 2
bash-3.2$ ls -lstr
112 -rwxr-xr-x  1 kennethroche  staff  55432 May  5 00:16 xdetach1
128 -rwxr-xr-x  1 kennethroche  staff  63768 May  5 00:25 xdetach-fil
  8 -rw-r-----@ 1 kennethroche  staff     22 May  5 00:26 file_one.txt
  8 -rw-r-----@ 1 kennethroche  staff     22 May  5 00:26 file_two.txt
```

- lambda functions

```
[capture](parameters) -> return_type {
    // function body
};
```

Capture Syntax	Meaning
`[=]`	Capture all outer variables **by value**
`[&]`	Capture all outer variables **by reference**
`[x]`	Capture only `x` **by value**
`[&x]`	Capture only `x` **by reference**

Lambda functions are **anonymous functions** (no name) introduced in **C++11**.

They allow **inline function definitions** — useful for short, throwaway functions.

Common use cases:

- Passing behavior to algorithms (`std::sort` , `std::for_each`)
- Capturing variables from the surrounding scope
- Launching threads

Kenneth J. Roche
University of Washington

- lambda functions

```
int x = 5, y = 7;
auto lambda = [=]() { return x + y; };  // captures by value
auto lambda2 = [&]() { x += 1; return x + y; };  // modifies x by reference
```

- Lambdas are ideal for defining thread tasks inline.
- No need to define a full function elsewhere.

```
#include <iostream>
#include <thread>

int main() {
    std::thread t([] {
        std::cout << "Running in a thread!" << std::endl;
    });
    t.join();  // Wait for thread to finish
    return 0;
}
```

- lambda functions
  - Captures are copied at the **moment the lambda is created**, not when the thread runs.
  - To modify `value` from within the lambda, use `[&]` **and be cautious of race conditions**.

```cpp
#include <iostream>
#include <thread>

int main() {
    int value = 42;

    std::thread t([value] {
        std::cout << "Captured value is: " << value << std::endl;
    });

    t.join();
    return 0;
}
```

Kenneth J. Roche
University of Washington

- lambda functions
  - Captures are copied at the **moment the lambda is created**, not when the thread runs.
  - To modify `value` from within the lambda, use `[&]` **and be cautious of race conditions**.

```cpp
#include <iostream>
#include <thread>

int main() {
    int shared = 100;

    std::thread t1([&] {
        shared += 1;
        std::cout << "Thread 1 incremented shared to " << shared << std::endl;
    });

    std::thread t2([&] {
        shared += 2;
        std::cout << "Thread 2 incremented shared to " << shared << std::endl;
    });

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

    std::cout << "Final shared value: " << shared << std::endl;
    return 0;
}
```

Corrected

```cpp
#include <iostream>
#include <thread>
#include <mutex>

int main() {
    int shared = 100;
    std::mutex mtx;

    std::thread t1([&] {
        std::lock_guard<std::mutex> lock(mtx);
        shared += 1;
        std::cout << "Thread 1 incremented shared to " << shared << std::endl;
    });

    std::thread t2([&] {
        std::lock_guard<std::mutex> lock(mtx);
        shared += 2;
        std::cout << "Thread 2 incremented shared to " << shared << std::endl;
    });

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

    std::cout << "Final shared value: " << shared << std::endl;
    return 0;
}
```

- templated lambda functions

```cpp
#include <iostream>

template<typename T>
auto add(T number) {
    return [number](T a, T b) -> T { return number * a + b; };
}

int main() {

    // assign the lambda for a specific number to a local lambda
    auto add_int = add<int>(3);
    auto add_double = add<double>(2.5);

    // Call the lambda function with different types
    int result1 = add_int(3, 4); // Result: number*3 + 4
    double result2 = add_double(2.5, 3.7); // Result: number*2.5 + 3.7

    std::cout << "lambda result of <int>    addition: " << result1 << std::endl;
    std::cout << "lambda result of <double> addition: " << result2 << std::endl;

    return 0;
}
```

```
bash-3.2$ g++ -std=c++17 -o xtmpl-lmbda cpp-lambda_3.cpp
bash-3.2$ ./xtmpl-lmbda
lambda result of <int>    addition: 13
lambda result of <double> addition: 9.95
bash-3.2$
```

- templated lambda functions

`template<typename T>` : This makes the `add` function **generic**, so it works for `int`, `double`, etc.

`add(T number)` : Takes one argument `number` of type `T`.

`return [...](...)` : This returns a **lambda function** that:

- Captures `number` by value.
- Accepts two arguments `a` and `b` of type `T`.
- Returns `number * a + b`.

```cpp
#include <iostream>

template<typename T>
auto add(T number) {
    return [number](T a, T b) -> T { return number * a + b; };
}

int main() {

    // assign the lambda for a specific number to a local lambda
    auto add_int = add<int>(3);
    auto add_double = add<double>(2.5);

    // Call the lambda function with different types
    int result1 = add_int(3, 4); // Result: number*3 + 4
    double result2 = add_double(2.5, 3.7); // Result: number*2.5 + 3.7

    std::cout << "lambda result of <int>    addition: " << result1 << std::endl;
    std::cout << "lambda result of <double> addition: " << result2 << std::endl;

    return 0;
}
```

```cpp
auto add_int = add<int>(3);         // number = 3
auto add_double = add<double>(2.5); // number = 2.5
```

`add_int(3, 4)` becomes `3*3 + 4 = 13`

`add_double(2.5, 3.7)` becomes `2.5*2.5 + 3.7 = 6.25 + 3.7 = 9.95`

```
bash-3.2$ g++ -std=c++17 -o xtmpl-lmbda cpp-lambda_3.cpp
bash-3.2$ ./xtmpl-lmbda
lambda result of <int>    addition: 13
lambda result of <double> addition: 9.95
bash-3.2$
```

Kenneth J. Roche
University of Washington

End Lecture 16

Kenneth J. Roche
University of Washington
AMATH 483 / 583 High Performance Scientific Computing
Spring Quarter 2025