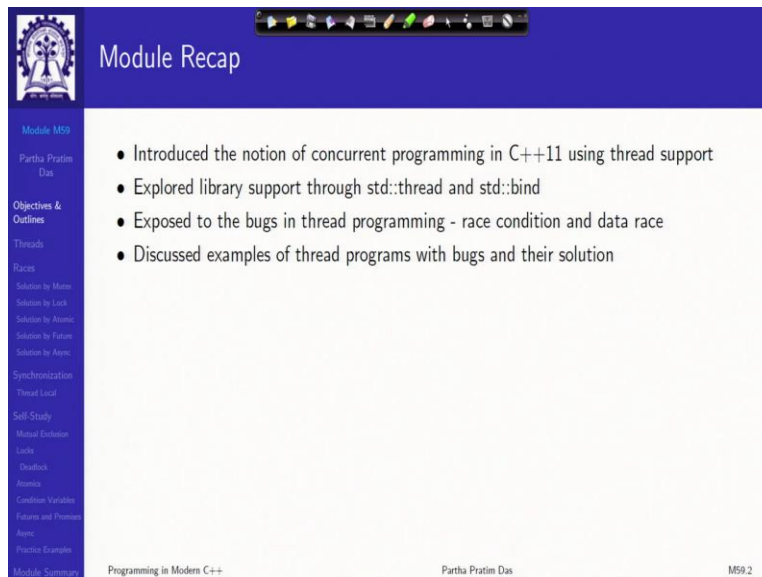


Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture – 59
C++ and Beyond: Concurrency: Part 2

Welcome to programming in modern C++, we are in week 12. And I am going to discuss module 59.

(Refer Slide Time: 00:38)



Module Recap

- Introduced the notion of concurrent programming in C++11 using thread support
- Explored library support through `std::thread` and `std::bind`
- Exposed to the bugs in thread programming - race condition and data race
- Discussed examples of thread programs with bugs and their solution

Programming in Modern C++ Partha Pratim Das M59.2

In the last module, I introduced the concept of concurrent programming in C++11 using the thread support that has been provided in the C++11 standard library. So, this makes the multi threaded programming which was earlier to be done by external third party libraries like the POSIX library or the boost library, much easier to do and a lot more standardized. We have explored the library support through `std::thread`, `bind`.

And we have seen that a naive coding offered sequential program into a multi threaded program will lead to certain bugs. A major category of bugs falling in race condition and data race we have seen, I discussed examples of trade programs with bugs and that solution.

(Refer Slide Time: 01:39)

Module Objectives

Module M59
Partha Pratim Das

Objectives & Outlines

- To understand synchronization issues in multi-thread programming in C++
- To study various synchronization mechanisms through example
- To self-study the details of synchronization mechanisms:
 - *Mutex*
 - *Lock*
 - *Atomics*
 - *Condition Variable*
 - *Future and Promises*
 - *Async*

Threads
Races
Solution by Mutex
Solution by Lock
Solution by Atomic
Solution by Future
Solution by Async
Synchronization
Thread Local
Self-Study
Mutex Eratosthenes
Locks
Deadlock
Atomic
Condition Variable
Future and Promises
Async
Practice Examples
Module Summary

Programming in Modern C++ Partha Pratim Das M59.3

We will continue on that to give you some more glimpses of the issues in multi threaded programming particularly the synchronization issues. And using the same example that we did in the last module, we will show more synchronization mechanisms as examples. Since concurrency support the threading support is a very, very large subject, we will not have time to discuss each one of the important features of the library in detail.

So, what I have done?? I have prepared a set of slides at the end of this presentation under the self-study banner. So, that you can study some details of specific synchronization mechanisms yourself and also practice examples that are given at the end.

(Refer Slide Time: 02:40)

Module Outline

- 1 Threads
- 2 Race Condition and Data Race
 - Solution by Mutex
 - Solution by Lock
 - Solution by Atomic
 - Solution by Future
 - Solution by Async
- 3 Synchronization
 - Thread Local
- 4 Self-Study
 - Mutual Exclusion
 - Locks
 - Deadlock
 - Atomics
 - Sync: Condition Variables
 - Sync: Futures and Promises
 - Async
 - Practice Examples
- 5 Module Summary

Programming in Modern C++ Parth Pratim Das M59.4

This is the overall outline and we will continue from where we had left.

(Refer Slide Time: 02:51)

Threads

Sources:

- C++11 - the new ISO C++ standard: Threads, Stroustrup, 2016
- C++11 Standard Library Extensions — Concurrency: Threads, isocpp
- std::thread, cppreference
- Concurrency memory model, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- A tutorial on modern multithreading and concurrency in C++, 2020
- C++11 Multi-threading Tutorials: Parts 1-8, thisPointer
 - C++11 Multithreading – Part 1 : Three Different ways to Create Threads
- C++20 Concurrency: Parts 1-3, Gajendra Gulgulia, 2021
 - C++20 Concurrency: Part 1: synchronized output stream
 - C++20 Concurrency: Part 2: jthreads
 - C++20 Concurrency: Part 3: request_stop and stop_token for std::jthread

Threads

Programming in Modern C++ Parth Pratim Das M59.5

Just a quick recap from the last module.

(Refer Slide Time: 02:52)

The slide is titled "Threads" and is part of a presentation on "Programming in Modern C++". It features a blue header with the title and a small video inset of the presenter, Parthiv Das. The main content area is white and contains a list of sources for further reading on C++ threads and concurrency. The slide also includes a navigation sidebar on the left and a footer with the presenter's name and the module number M59.5.

Threads

Sources:

- C++11 - the new ISO C++ standard: Threads, Stroustrup, 2016
- C++11 Standard Library Extensions — Concurrency: Threads, isocpp
- std::thread, cppreference
- Concurrency memory model, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- A tutorial on modern multithreading and concurrency in C++, 2020
- C++11 Multi-threading Tutorials: Parts 1-8, thisPointer
 - C++11 Multithreading – Part 1 : Three Different ways to Create Threads
- C++20 Concurrency: Parts 1-3, Gajendra Gulguia, 2021
 - C++20 Concurrency: Part 1: synchronized output stream
 - C++20 Concurrency: Part 2: jthreads
 - C++20 Concurrency: Part 3: request_stop and stop_token for std::jthread

Threads

Programming in Modern C++ Parthiv Das M59.5

We showed that thread is a lightweight process and we have learned the basic trade operations using the thread component of the standard library, that is creating a thread, passing parameters to a thread directly or by `std::bind`, returning the result from a thread directly or by `std::bind`, joining threads and so on. And we have also observed race condition and data race in the multi threaded program.

(Refer Slide Time: 03:23)

The slide is titled "Race Condition and Data Race" and is part of a presentation on "Programming in Modern C++". It features a blue header with the title and a small video inset of the presenter, Parthiv Das. The main content area is white and contains the title "Race Condition and Data Race" in red. The slide also includes a navigation sidebar on the left and a footer with the presenter's name and the module number M59.7.

Race Condition and Data Race

Race Condition and Data Race

Programming in Modern C++ Parthiv Das M59.7

Now, just again to recap that what is race condition or data race?

(Refer Slide Time: 03:29)

Race Condition and Data Race

- We often talk about bugs in multi-threading:
 - *Race Condition*
 - *Data Race*
- Are they same?
 - No, they are not
 - They are not a subset of one another
 - They are also neither the necessary, nor the sufficient condition for one another
- **Race Condition:** A race condition is a semantic error
 - A race condition is a situation, in which the result of an operation depends on the interleaving of certain individual operations
 - Many race conditions can be caused by data races, but this is not necessary
- **Data Race:** A data race occurs when 2 instructions from different threads access the same memory location without synchronization
 - A data race is a situation, in which at least two threads access a shared variable at the same time. At least one thread tries to modify the variable.
 - The discovery of data race can be automated
- We take examples to illustrate both

Module M59
Partha Pratim Das
Objectives & Outlines
Threads
Races
Solution by Mutex
Solution by Lock
Solution by Atomic
Solution by Future
Solution by Await
Synchronization
Thread Local
Self-Study
Mutex
Locks
Feedback
Assess
Condition Variables
Future and Promise
Async
Practice Examples
Module Summary

Programming in Modern C++ Partha Pratim Das M59.8

These are kind of synchronization problems. So, they are very closely related though they are not exactly the same thing. As you have seen a race condition is a semantic error it is a situation in which the result of operation depends on the interleaving of certain individual operations which may occur in indeterminate order in multiple threads.

And a data race particularly occurs on I mean the condition where the at least two instructions are trying to come in from different threads or trying to access the same memory location without synchronization. And at least one of them is trying to write that data race rises.

(Refer Slide Time: 04:16)

Example 1

- Let us write a simple program to compute:

$$\sum_{i=1}^{20} i^2 = \frac{20 \times (20 + 1) \times (2 \times 20 + 1)}{6} = 2870$$

- Assuming that `x*x` is a heavy computation (fake it!) we developed a simple multi-threaded program for the above:
 - Spawn 20 threads
 - Each thread computes `square` for a distinct value
 - The accumulated result is available after the threads join
- We added random delay and repeated run support to setup scenarios for race conditions to be observed. We observed that bugs exist
- We have also discussed two fixes – by `mutex` and by `atomic` which we will recap here
- We also discuss other solutions by `lock`, `future`, and `async`

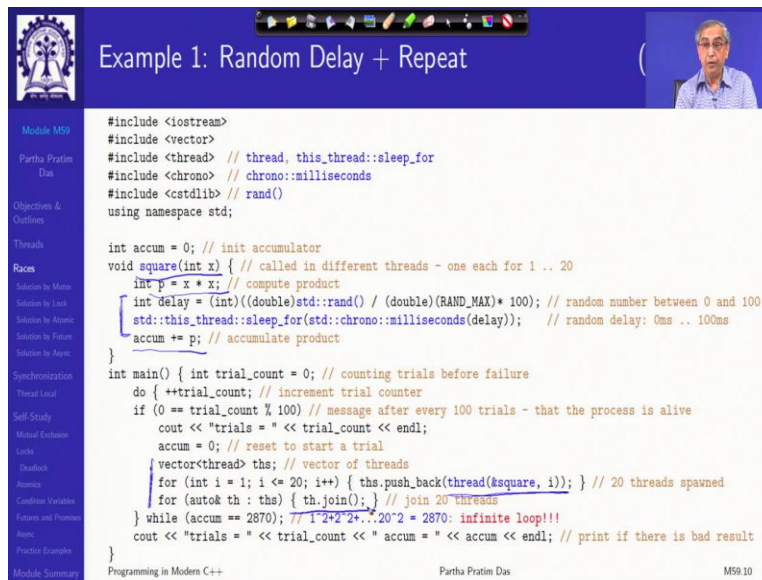
Programming in Modern C++ Parthiv Pratin Das M59 9

Now, we will continue with the same example which was to find the sum of squares of 1 to 20 and we had assumed that `x multiplied by x` is a heavy operation to simulate that we have also modeled a random delay after the computation of this product in accumulating the results. And to be able to repeat this, the thread bugs are kind of indeterminate ones. And they may show up in 1 run and they may not show up in 100 other runs.

So, to get to a bug which is particularly arising from the synchronization, we will typically need to run the program several times. And therefore, we also created a setup where we could keep on repeating the runs of the program and check whether we get the correct result. If we do not, then we know that we have a bug. If we do not get an error, then we have a better confidence, but we can never be sure just by this observation that there is no error.

So, we have what we have done is we have discussed two fixes for this problem, one using `mutex` and the other using `atomic`. And we will use, we will show a couple of more here after and of course, revise the earlier ones, revisit the earlier ones.

(Refer Slide Time: 05:53)



The screenshot shows a presentation slide with a blue header containing the title "Example 1: Random Delay + Repeat" and a small video feed of a speaker. The main content is a C++ code snippet. The code includes headers for `<iostream>`, `<vector>`, `<thread>`, `<chrono>`, and `<cstdlib>`. It defines a `square` function that computes x^2 and an accumulator. The `main` function runs a loop of trials, each containing 20 threads that compute squares with a random delay. A `while` loop continues until the accumulator reaches 2870, which is the sum of squares from 1 to 20. The code includes comments and a `while` loop that runs until the accumulator reaches 2870, with a comment indicating it's an "infinite loop".

```
#include <iostream>
#include <vector>
#include <thread> // thread, this_thread::sleep_for
#include <chrono> // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;

int accum = 0; // init accumulator
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay)); // random delay: 0ms .. 100ms
    accum += p; // accumulate product
}

int main() { int trial_count = 0; // counting trials before failure
do { ++trial_count; // increment trial counter
if (0 == trial_count % 100) // message after every 100 trials - that the process is alive
    cout << "trials = " << trial_count << endl;
    accum = 0; // reset to start a trial
    vector<thread> ths; // vector of threads
    for (int i = 1; i <= 20; i++) { ths.push_back(thread(&square, i)); } // 20 threads spawned
    for (auto& th : ths) { th.join(); } // join 20 threads
} while (accum == 2870); // 1^2+2^2+...20^2 = 2870: infinite loop!!!
cout << "trials = " << trial_count << " accum = " << accum << endl; // print if there is bad result
}
```

So, this is this was the version of the program, which we are trying to trying to synchronize. Here is you can see that there is a function square which takes and compute the square and accumulates and here is where we put a arbitrary amount of random delay to kind of simulate that the product operation is a heavy operation. And here is where we do multiple runs, this is the actual body of the program where we create an array of default area of threads.

And then put different threads with the associated square function, but different parameters to put them in the vector. So, that as soon as I put them they start working and they will get variable amount of delay amongst them, and then it will accumulate and this accumulation will be completed when all threads have joined. And we check whether that related result is equal to the as desired. And we have seen on several occasions that it fails, but synchronisation can help solve that problem.

(Refer Slide Time: 07:14)

Example 1: Solution by Mutex

Module M59
Partha Pratim Das

Objectives & Outlines
Threads
Races
Solution by Mutex
Solution by Lock
Solution by Atomic
Solution by Future
Solution by Async

Synchronization
Thread Local
Self-Study
Mutual Exclusion
Locks
Deadlock
Atomic
Condition Variables
Futures and Promises
Async
Practice Examples
Module Summary

Race Condition and Data Race: Example 1: Solution by Mutex

Programming in Modern C++ Partha Pratim Das M59.11

So, the first solution which we have seen is by doing a mutex.

(Refer Slide Time: 07:18)

Example 1: Solution by Mutex

Module M59
Partha Pratim Das

Objectives & Outlines
Threads
Races
Solution by Mutex
Solution by Lock
Solution by Atomic
Solution by Future
Solution by Async

Synchronization
Thread Local
Self-Study
Mutual Exclusion
Locks
Deadlock
Atomic
Condition Variables
Futures and Promises
Async
Practice Examples
Module Summary

- A mutex (mutual exclusion) allows us to encapsulate blocks of code that should only be executed in one thread at a time. Keeping the main function the same:

```
int accum = 0;
mutex accum_mutex; // mutex variable

void square(int x) {
    int temp = x * x;
    accum_mutex.lock(); // gets the lock on accum_mutex
    accum += temp;
    accum_mutex.unlock(); // release the lock on accum_mutex
}
```
- We try running the program repeatedly again and the problem should now be fixed
- The first thread that calls `lock()` gets the lock
- During this time, all other threads that call `lock()`, will wait at that line for the mutex to be unlocked. Creates a Critical Section
- It is important to introduce the variable `temp`, since we want the `x * x` calculations to be outside the lock-unlock block, otherwise we would be hogging the lock while we are running our heavy calculations

Programming in Modern C++ Partha Pratim Das M59.12

Example 1: Solution by Mutex

- A mutex (mutual exclusion) allows us to encapsulate blocks of code that should only be executed in one thread at a time. Keeping the main function the same:


```
int accum = 0;
mutex accum_mutex; // mutex variable

void square(int x) {
    int temp = x * x;
    accum_mutex.lock(); // gets the lock on accum_mutex
    accum += temp;
    accum_mutex.unlock(); // release the lock on accum_mutex
}
```
- We try running the program repeatedly again and the problem should now be fixed
- The first thread that calls `lock()` gets the lock
- During this time, all other threads that call `lock()`, will wait at that line for the mutex to be unlocked. Creates a *Critical Section*
- It is important to introduce the variable `temp`, since we want the `x * x` calculations to be outside the lock-unlock block, otherwise we would be hogging the lock while we are running our heavy calculations

Module M59
Partha Pratim Das
Objectives & Outlines
Threads
Races
Solution by Mutex
Solution by Lock
Solution by Atomic
Solution by Atom
Synchronization
Thread Local
Self-Study
Mutual Exclusion
Locks
Deadlock
Atomic
Condition Variables
Fetch and Process
Atomic
Practical Examples
Module Summary

Programming in Modern C++ Partha Pratim Das M59.12

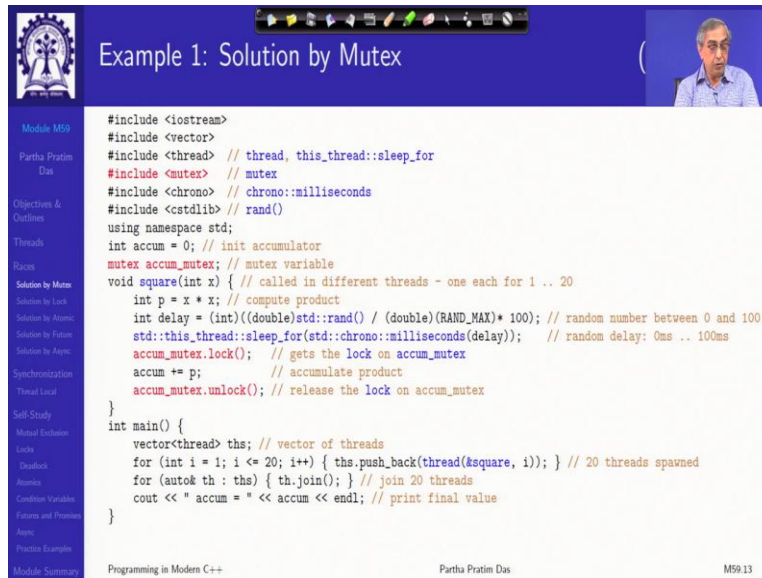
Mutex is like a gate. So, what it creates is what is known as a critical region. So, this is known as a critical section or a critical region. So, what it does is when a thread reaches this point, and assuming that no other thread has reached that point, then it will acquire that lock. So, it can acquire that lock provided no other thread is currently acquired it. So, once it has acquired the lock, it can proceed and perform this addition.

But once, one thread has acquired the lock, no other thread which has reached this point, will be able to acquire the lock. They will have to wait at that point. So, they will be waiting at this point there is no kill. The thread which was having the lock executes the unlock releases it, so that out of the waiting threads, one thread will actually add the lock and will be able to proceed.

So, that simply means that even though square is running on multiple threads, this part of the function square, which is computing the square and of course, it will, we will put that random delay to make it a heavy task. We will be done in concurrent fashion all threads can do it at the same time at different times and so on. Whereas, this part the critical section part will necessarily be serialized.

That is at any point of time only one thread will be able to add the product of x into x to the accumulator. So, this will ensure that the problem that we had seen earlier that the old value of an accumulator is being read and being updated will not arise.

(Refer Slide Time: 09:15)



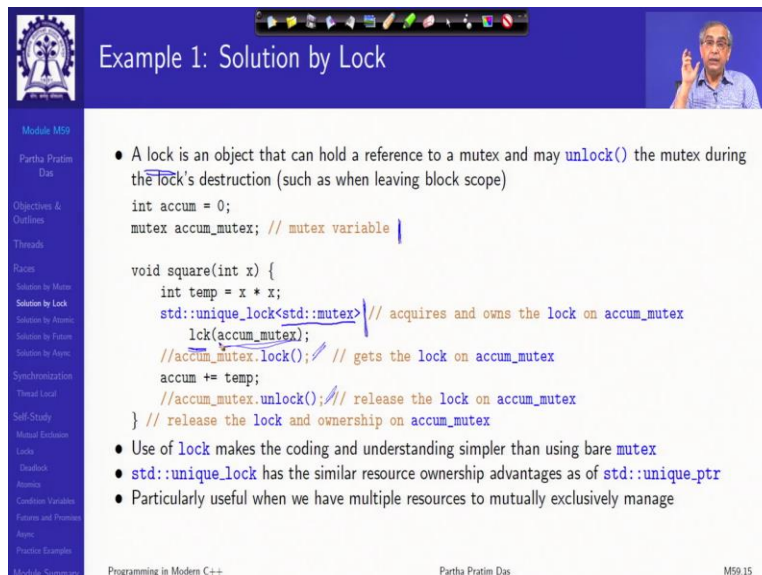
Example 1: Solution by Mutex

```
#include <iostream>
#include <vector>
#include <thread> // thread, this_thread::sleep_for
#include <mutex> // mutex
#include <chrono> // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;
int accum = 0; // init accumulator
mutex accum_mutex; // mutex variable
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay)); // random delay: 0ms .. 100ms
    accum_mutex.lock(); // gets the lock on accum_mutex
    accum += p; // accumulate product
    accum_mutex.unlock(); // release the lock on accum_mutex
}
int main() {
    vector<thread> ths; // vector of threads
    for (int i = 1; i <= 20; i++) { ths.push_back(thread(&square, i)); } // 20 threads spawned
    for (auto& th : ths) { th.join(); } // join 20 threads
    cout << " accum = " << accum << endl; // print final value
}
```

Programming in Modern C++ Partha Pratim Das M59.13

And with that, we have seen the modified solution, the safe solution, whereby red I have shown the changes that you make to the central thread program, of course, I have not shown the reputation part because that's basically an experimental setup, but this is a program which will work correctly.

(Refer Slide Time: 09:34)



Example 1: Solution by Lock

- A lock is an object that can hold a reference to a mutex and may `unlock()` the mutex during the lock's destruction (such as when leaving block scope)

```
int accum = 0;
mutex accum_mutex; // mutex variable

void square(int x) {
    int temp = x * x;
    std::unique_lock<std::mutex> lck(accum_mutex); // acquires and owns the lock on accum_mutex
    //accum_mutex.lock(); // gets the lock on accum_mutex
    accum += temp;
    //accum_mutex.unlock(); // release the lock on accum_mutex
} // release the lock and ownership on accum_mutex
```

- Use of `lock` makes the coding and understanding simpler than using bare `mutex`
- `std::unique_lock` has the similar resource ownership advantages as of `std::unique_ptr`
- Particularly useful when we have multiple resources to mutually exclusively manage

Programming in Modern C++ Partha Pratim Das M59.15

Example 1: Solution by Lock

- A lock is an object that can hold a reference to a mutex and may `unlock()` the mutex during the lock's destruction (such as when leaving block scope)

```

int accum = 0;
mutex accum_mutex; // mutex variable

void square(int x) {
    int temp = x * x;
    std::unique_lock<std::mutex> lck(accum_mutex); // acquires and owns the lock on accum_mutex
    lck(accum_mutex);
    //accum_mutex.lock(); // gets the lock on accum_mutex
    accum += temp;
    //accum_mutex.unlock(); // release the lock on accum_mutex
    // release the lock and ownership on accum_mutex
}

```

- Use of `lock` makes the coding and understanding simpler than using bare `mutex`
- `std::unique_lock` has the similar resource ownership advantages as of `std::unique_ptr`
- Particularly useful when we have multiple resources to mutually exclusively manage

Programming in Modern C++ Partha Pratim Das M59.15

Example 1: Solution by Mutex

- A mutex (mutual exclusion) allows us to encapsulate blocks of code that should only be executed in one thread at a time. Keeping the main function the same:

```

int accum = 0;
mutex accum_mutex; // mutex variable

void square(int x) {
    int temp = x * x;
    accum_mutex.lock(); // gets the lock on accum_mutex
    accum += temp;
    accum_mutex.unlock(); // release the lock on accum_mutex
}

```

- We try running the program repeatedly again and the problem should now be fixed
- The first thread that calls `lock()` gets the `lock`
- During this time, all other threads that call `lock()`, will wait at that line for the mutex to be unlocked. Creates a *Critical Section*
- It is important to introduce the variable `temp`, since we want the `x * x` calculations to be outside the lock-unlock block, otherwise we would be hogging the lock while we are running our heavy calculations

Programming in Modern C++ Partha Pratim Das M59.12

Now, we will introduce a modification or a refinement of this mutex solution using something what is known as a lock. Lock works in this way that again suppose you have a mutex variable on which you could have done `mutex.lock()` `mutex dot`, I mean, `mutex.lock()` and `mutex.unlock()` as you have been doing here.

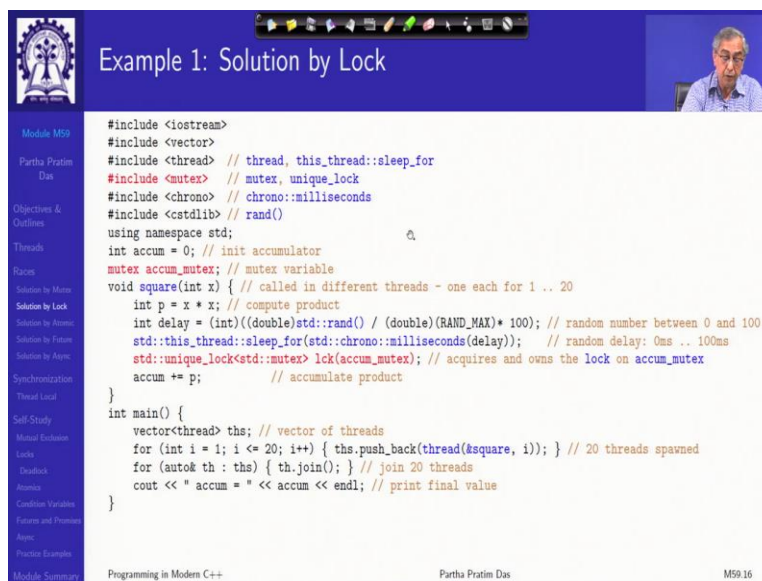
But instead of doing that, what you do is you do this. That is you say `std::unique_lock`, which is available in the `mutex` component and pass a template parameter `mutex`, then this is your lock variable, and this is the mutex that you are locking. So, what you say is basically, a `unique_lock` is a template function which takes an object of a certain type and locks it. Here, it is taking an object of a mutex type.

And that object is the mutex that I have created, this acute mutex. And it locks it. The advantage of doing this is so as you do that, you necessarily do not have this and you do not have this. But what happens when such a lock goes to the end of its scope, this is the end of the scope, then it automatically it is destructor of that object that is a structure of lck will automatically get called. The destructor will actually unlock the mutex object that you have locked.

So, you can see that the advantage here is you get the semantics of the automatic variables here in terms of locking. So, you get the same critical section, but you do not have to parenthesize it. Because you might just forget, if there are many at different places, you might just forget to unlock a particular mutex when it is done. So, this relieves you of that. This will remind you of if you recall, smart pointers, we had unique pointers for the same purpose where the unique pointer was taking a raw pointer and taking ownership of it.

And when the unique pointer that goes out of scope, its destructor automatically destroys the object pointed to by that pointer and disappears. So, exactly in the same way unit lock achieves a resource management for the locks, which makes programming far more easier.

(Refer Slide Time: 12:28)



```
#include <iostream>
#include <vector>
#include <thread> // thread, this_thread::sleep_for
#include <mutex> // mutex, unique_lock
#include <chrono> // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;

int accum = 0; // init accumulator
mutex accum_mutex; // mutex variable
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay)); // random delay: 0ms .. 100ms
    std::unique_lock<std::mutex> lck(accum_mutex); // acquires and owns the lock on accum_mutex
    accum += p; // accumulate product
}

int main() {
    vector<thread> ths; // vector of threads
    for (int i = 1; i <= 20; i++) { ths.push_back(thread(&square, i)); } // 20 threads spawned
    for (auto& th : ths) { th.join(); } // join 20 threads
    cout << " accum = " << accum << endl; // print final value
}
```

So, using the lock, if we right now, then the program becomes actually much simpler, we will still need to define the mutex variable, but it is just one line of unique lock that before the accumulation of the product which solves the problem. So, this is the second way of doing this.

(Refer Slide Time: 12:49)

Example 1: Solution by Atomic

Module M59
Partha Pratim Das

Objectives & Outlines
Threads
Races
Solution by Mutex
Solution by Lock
Solution by Atomic
Solution by Future
Solution by Async
Synchronization
Thread Local
Self-Study
Mutual Exclusion
Locks
Deadlock
Atomic
Condition Variables
Futures and Promises
Async
Practice Examples
Module Summary

Race Condition and Data Race: Example 1: Solution by Atomic

Programming in Modern C++ Partha Pratim Das M59.17

You could make use of lock which is based on the, on the mutex.

(Refer Slide Time: 12:55)

Example 1: Solution by Atomic

Module M59
Partha Pratim Das

Objectives & Outlines
Threads
Races
Solution by Mutex
Solution by Lock
Solution by Atomic
Solution by Future
Solution by Async
Synchronization
Thread Local
Self-Study
Mutual Exclusion
Locks
Deadlock
Atomic
Condition Variables
Futures and Promises
Async
Practice Examples
Module Summary

- With Mutex / Lock the problem gets fixed. The program does not produce a wrong result even after 6000+ trials
- Interestingly, **C++11** offers even nicer abstractions to solve this problem. For instance, the **atomic** container:

```
#include <atomic>
atomic<int> accum(0); // makes accum and initializes to 0
void square(int x) {
    accum += x * x;
}
```

incl accum = 0

- We do not need to introduce temp here, since $x * x$ will be evaluated before handed off to **accum**, so it will be outside the atomic event
- However, we will continue to show the solution using the temporary

Programming in Modern C++ Partha Pratim Das M59.18

Example 1: Solution by Atomic

- With Mutex / Lock the problem gets fixed. The program does not produce a wrong result even after 6000+ trials
- Interestingly, **C++11** offers even nicer abstractions to solve this problem. For instance, the **atomic** container:


```
#include <atomic>

atomic<int> accum(0); // makes accum and initializes to 0

void square(int x) {
    accum += x * x;
}
```
- We do not need to introduce temp here, since $x * x$ will be evaluated before handed off to **accum**, so it will be outside the atomic event
- However, we will continue to show the solution using the temporary

Programming in Modern C++ | Partha Pratim Das | M59.18

The third solution is what we have seen in the last module also that you have a component atomic by which any variable which needs to be updated in a critical section or needs to be updated safely by multiple threads can be declared as atomic. So, I am, I am reexplaining that so this is a component. And what I am saying is accumulate, accum is a variable of type int. So, you are making it an atomic int initialized with value 0.

That is instead of doing `int accum`, which makes it a global, simple global variable, anybody can change anytime, you make it an atomic in terms of. The advantage is after that you just do whatever you had been doing. What happens is when this part is happening, the threads are allowed to go concurrently. But when you try to update `accum`, the atomic behavior that is defined in the library will make sure that threads are properly serialized.

And this update happens as an atomic operation that is when one thread is doing the update that it has read the value of the accumulator and is adding the product to it and getting the final value. During this time, no other thread will be able to do that. So, this atomicity is what is important and which comes very handy in terms of simple synchronization problems.

(Refer Slide Time: 14:36)

The slide displays a C++ program that uses a mutex to ensure thread safety. The code includes headers for `<iostream>`, `<vector>`, `<thread>`, `<mutex>`, `<chrono>`, and `<cstdlib>`. It defines a `square` function that takes an integer `x` and returns `x * x`. The `main` function creates a vector of 20 threads, each of which calls `square` on a value from 1 to 20. A `mutex` named `accum_mutex` is used to protect the `accum` variable, which is updated by each thread. The program outputs the final value of `accum`.

```
#include <iostream>
#include <vector>
#include <thread> // thread, this_thread::sleep_for
#include <mutex> // mutex, unique_lock
#include <chrono> // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;
int accum = 0; // init accumulator
mutex accum_mutex; // mutex variable
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay)); // random delay: 0ms .. 100ms
    std::unique_lock<std::mutex> lck(accum_mutex); // acquires and owns the lock on accum_mutex
    accum += p; // accumulate product
}
int main() {
    vector<thread> ths; // vector of threads
    for (int i = 1; i <= 20; i++) { ths.push_back(thread(&square, i)); } // 20 threads spawned
    for (auto& th : ths) { th.join(); } // join 20 threads
    cout << " accum = " << accum << endl; // print final value
}
```

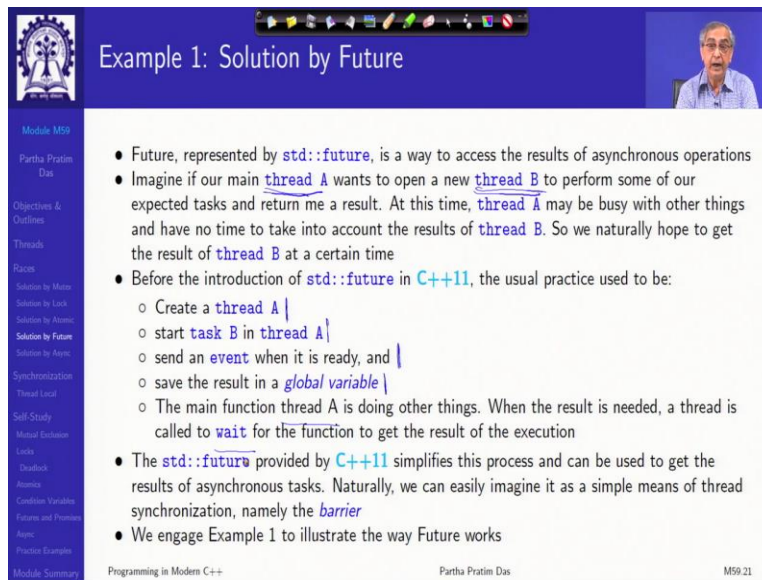
So, again with this, there is almost no change in your earlier unsafe program. All that you need is replaced the global declaration with the declaration of atomic int for `accum` with initialization 0.

(Refer Slide Time: 14:55)

The slide features a red heading: **Race Condition and Data Race: Example 1: Solution by Future**. The rest of the slide content is currently blank.

We look at a new solution using what is known as future, future and promise.

(Refer Slide Time: 15:02)



The slide is titled "Example 1: Solution by Future" and features a small video inset of the presenter in the top right corner. The main content is a list of bullet points explaining the concept of `std::future` in C++11. The slide also includes a navigation sidebar on the left and footer information at the bottom.

- Future, represented by `std::future`, is a way to access the results of asynchronous operations
- Imagine if our main `thread A` wants to open a new `thread B` to perform some of our expected tasks and return me a result. At this time, `thread A` may be busy with other things and have no time to take into account the results of `thread B`. So we naturally hope to get the result of `thread B` at a certain time
- Before the introduction of `std::future` in C++11, the usual practice used to be:
 - Create a `thread A` |
 - start `task B` in `thread A` |
 - send an `event` when it is ready, and |
 - save the result in a `global variable` |
 - The main function `thread A` is doing other things. When the result is needed, a thread is called to `wait` for the function to get the result of the execution
- The `std::future` provided by C++11 simplifies this process and can be used to get the results of asynchronous tasks. Naturally, we can easily imagine it as a simple means of thread synchronization, namely the `barrier`
- We engage Example 1 to illustrate the way Future works

Navigation sidebar (left):
Module M59
Partha Pratim Das
Objectives & Outlines
Threads
Races
Solution by Mutex
Solution by Lock
Solution by Atomic
Solution by Future
Solution by Atomic
Synchronization
Thread Local
Self-Study
Manual Execution
Locks
Deadlock
Atomic
Condition Variables
Futures and Promises
Atoms
Practice Examples
Module Summary

Footer (bottom):
Programming in Modern C++ | Partha Pratim Das | M59.21

So, future is kind of a way to access the result of an asynchronous operation. You know, threads are doing things on their own. So, there is no as such, they are not synchronized, so they are doing things, someone is computing, someone else need to use that. So, if you have to access the result of an asynchronous operation, let us say, thread A has a task to do, which it has given to thread B. And meanwhile, thread A is doing something else.

So, thread B is to perform that task and return the result. Now, thread A is not waiting for the result, like `join thread B`, it is not waiting like that, but it is doing its own work. And it will like to get the result in its own sweet time after B has computed it. So, what you typically do, in this case is I mean before without using future, you create the thread, create the task B in thread A, so spawn that thread. Send an event when the, when it is ready, and save the result in a global variable.

Again, bad programming using global variables, but nothing can be done. And then the thread A is doing other things. And when the result need it, it is called to wait function so that if thread B has finished, then A gets that result. Now that whole functionality is now compactly given by future, future and promise they go hand in hand.

(Refer Slide Time: 16:53)

Example 1: Solution by Future

```
#include <iostream>
#include <vector>
#include <thread> // thread, this_thread::sleep_for
#include <future> // future
#include <chrono> // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;
int accum = 0; // define accumulator
void square(future<int>& fut) { // called in different threads - one each for 1 .. 20
    int x = fut.get(); // get parameter from future
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay)); // random delay: 0ms .. 100ms
    accum += p; // accumulate product
}
int main() {
    vector<promise<int>> vp; //promises*/ vector<future<int>> vf; /*futures*/ vector<thread> vt;
    for (int i = 0; i < 20; i++) {
        vp.push_back(promise<int>()); // vector of promise objects
        vf.push_back(vp[i].get_future()); // vector of engaged future objects from promise objects
        vt.push_back(thread(&square, ref(vf[i]))); // vector of threads - pass future object
        vp[i].set_value(i+1); // fulfil promise with needed value
    }
    for (auto& t : vt) { t.join(); } // join 20 threads
    cout << " accum = " << accum << endl; // print final value
}
```

Programming in Modern C++ Partha Pratim Das M59.22

Example 1: Solution by Future

```
#include <iostream>
#include <vector>
#include <thread> // thread, this_thread::sleep_for
#include <future> // future
#include <chrono> // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;
int accum = 0; // define accumulator
void square(future<int>& fut) { // called in different threads - one each for 1 .. 20
    int x = fut.get(); // get parameter from future
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay)); // random delay: 0ms .. 100ms
    accum += p; // accumulate product
}
int main() {
    vector<promise<int>> vp; //promises*/ vector<future<int>> vf; /*futures*/ vector<thread> vt;
    for (int i = 0; i < 20; i++) {
        vp.push_back(promise<int>()); // vector of promise objects
        vf.push_back(vp[i].get_future()); // vector of engaged future objects from promise objects
        vt.push_back(thread(&square, ref(vf[i]))); // vector of threads - pass future object
        vp[i].set_value(i+1); // fulfil promise with needed value
    }
    for (auto& t : vt) { t.join(); } // join 20 threads
    cout << " accum = " << accum << endl; // print final value
}
```

Programming in Modern C++ Partha Pratim Das M59.22

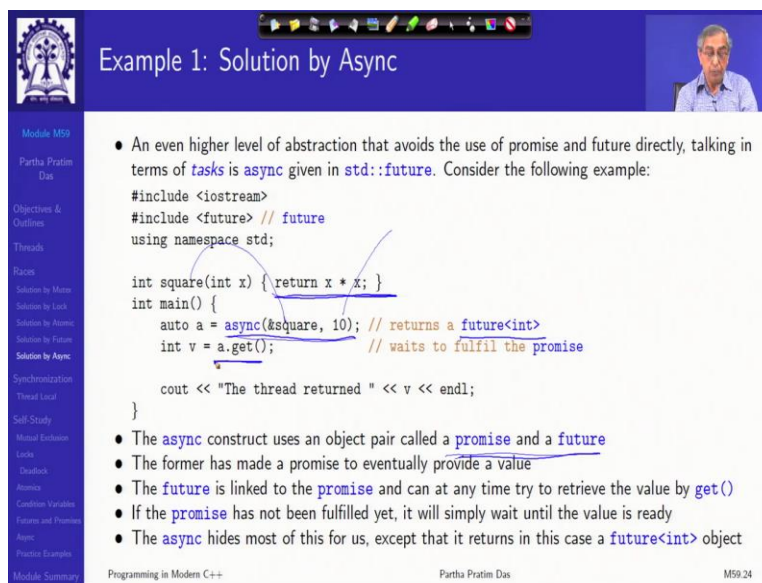
So, what it does, it basically initially creates a promise that I will give you, I will give you this in future. Then from the promise you get a future, you get a handle to a future. And that future is what you pass to your task, you pass that future to the task. And in the task, you can, you can obviously do a get to get the actual value that you have given the future to do.

Then your task, then you create this thread with that particular task that you want to do the square, with the future that you have given it. And after creating that, then in the promise, you set the value that you want to give to the future. So, you promise that you will give something to the

future. And according to that future is ready to take up the work. So, here as soon as you create that, the thread will start, but it will not be able to go forward.

Because the promised value has not been given yet. So, when you give that value in the in the promise by setting it, then it gets that and executes. And then threads go as they are. And they join at the end. And you have the result. Since this this future and promise will make sure that this synchronization is happening. It is it is somewhat complicated to think about, but that is a another nice way.

(Refer Slide Time: 18:41)



The slide, titled "Example 1: Solution by Async", features a blue header with a logo on the left and a small video inset of a man on the right. The main content area is white with blue text for code and bullet points. A vertical navigation menu is on the left side of the slide. The code block shows a C++ program using `std::async` and `std::future`. The first bullet point explains that `std::async` is a higher-level abstraction. The second bullet point describes the `async` construct as a pair of `promise` and `future`. The third bullet point notes that the `future` is linked to the `promise` and can retrieve the value via `get()`. The fourth bullet point states that if the `promise` is not fulfilled, the `future` will wait. The fifth bullet point mentions that `async` hides these details and returns a `future<int>` object.

```
#include <iostream>
#include <future> // future
using namespace std;

int square(int x) { return x * x; }
int main() {
    auto a = async(&square, 10); // returns a future<int>
    int v = a.get();           // waits to fulfil the promise

    cout << "The thread returned " << v << endl;
}
```

- An even higher level of abstraction that avoids the use of promise and future directly, talking in terms of *tasks* is `async` given in `std::future`. Consider the following example:
- The `async` construct uses an object pair called a `promise` and a `future`
- The former has made a promise to eventually provide a value
- The `future` is linked to the `promise` and can at any time try to retrieve the value by `get()`
- If the `promise` has not been fulfilled yet, it will simply wait until the value is ready
- The `async` hides most of this for us, except that it returns in this case a `future<int>` object

The fifth way to solve this problem is actually to package future and promise into something of a higher abstraction called `async`. What `async` does is it you have seen that you need a pair of promise and future to solve the problem. So, the question is if we always need that pair, why not define a high level object which makes the pair itself. So, you do not have to write all of these? Yes, the `async` will do all that pair.

So, I have the task. I do not pass any future tweet, I pass whatever I was passing and I return the value whatever I was returning. And I just instead of creating thread, I just created `async` object with the task and the parameter. This will return a future by itself. Because it is embodied, this will create promise internally future set that value do all that.

And then it will wait for the promise to be fulfilled, that is get. So, when you get that value, you have the actual value. So, that is that is the basic async operation, which is much simpler than using future and promise.

(Refer Slide Time: 20:09)

Example 1: Solution by Async

```
#include <iostream>
#include <vector>
#include <thread> // thread, this_thread::sleep_for
#include <future> // future
#include <chrono> // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;

int square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay)); // random delay: 0ms .. 100ms
    return p;
}

int main() {
    int accum = 0; // define accumulator
    vector<future<int>> fts; // vector of future objects
    for (int i = 1; i <= 20; i++) { fts.push_back(async(&square, i)); } // 20 future objects
    for (auto& ft : fts) { accum += ft.get(); } // wait to get value from future and accumulate
    cout << "accum = " << accum << endl; // print final value
}
```

- Works fine. Does not produce a wrong result even after 30000+ trials

Programming in Modern C++ Partha Pratim Das M59.25

Example 1: Solution by Async

```
#include <iostream>
#include <vector>
#include <thread> // thread, this_thread::sleep_for
#include <future> // future
#include <chrono> // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;

int square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); // random number between 0 and 100
    std::this_thread::sleep_for(std::chrono::milliseconds(delay)); // random delay: 0ms .. 100ms
    return p;
}

int main() {
    int accum = 0; // define accumulator
    vector<future<int>> fts; // vector of future objects
    for (int i = 1; i <= 20; i++) { fts.push_back(async(&square, i)); } // 20 future objects
    for (auto& ft : fts) { accum += ft.get(); } // wait to get value from future and accumulate
    cout << "accum = " << accum << endl; // print final value
}
```

- Works fine. Does not produce a wrong result even after 30000+ trials

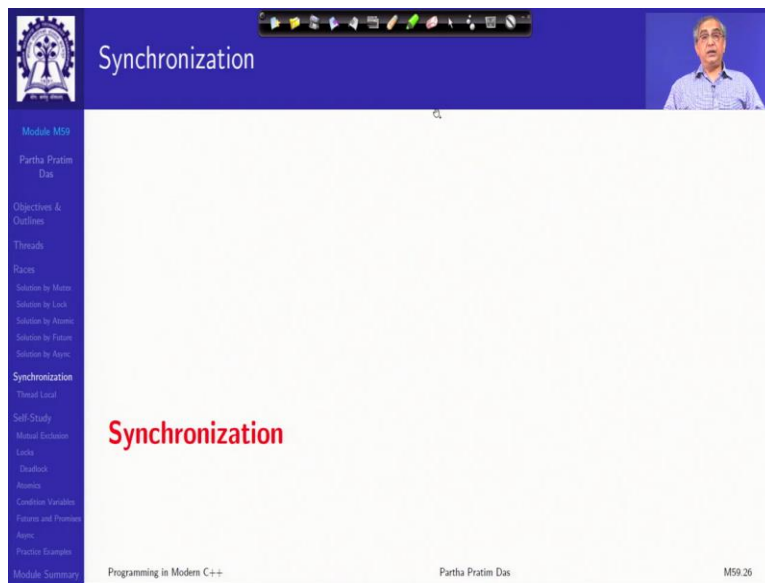
Programming in Modern C++ Partha Pratim Das M59.25

So, if we have to use this, now we have an accumulator again. And we have a, I have put the accumulator as a part of main instead of global, because my task function square, no more uses the accumulator. That task function is not adding it to the accumulator, because that is the synchronizable part. That is not the concurrent part. So, I create a vector of futures. And I do push back that, what did I say?

That async will return a future object. So, if I do async, the task function and the parameter it will give me a future object, so I put it to the vector. So, I have so many 20 future objects, and for these 20 future objects, each one of them have to complete. And as they complete, I add them and that I do sequential. This addition, I do sequentially here by this range for.

So, I do ft, that that future object, I do a get, and I wait, if it has not finished, and as soon as I get I add it to the accumulator. So, this happens one after the other. So, this part is serialized as I wanted, and I get a direct result.

(Refer Slide Time: 21:38)



So, I have shown you multiple different ways to address the basic synchronization problem of race condition, data races.

(Refer Slide Time: 21:48)

The slide is titled "Synchronization Errors: Symptoms and Causes" and features a list of five synchronization errors. Each error is accompanied by a brief description of its symptoms. The slide also includes a navigation menu on the left and a small video inset of the presenter in the top right corner.

Error	Symptoms
Conflicting access to shared memory	<ul style="list-style-type: none">one thread begins an operation on shared memory, is suspended, and leaves that memory region incompletely transformeda second thread is activated and accesses the shared memory in the corrupted state, causing errors in its operation and potentially errors in the operation of the suspended thread when it resumes
Race Conditions	<ul style="list-style-type: none">correct operation depends on the order of completion of two or more independent activitiesthe order of completion is not deterministic
Deadlock	<ul style="list-style-type: none">two or more tasks each own resources needed by the other preventing either one from running so neither ever completes and never releases its resource
Starvation	<ul style="list-style-type: none">a high priority thread dominates CPU resources, preventing lower priority threads from running often enough or at all
Priority inversion	<ul style="list-style-type: none">a low priority task holds a resource needed by a higher priority task, blocking it from running

So, just to give you a broader idea, let me take you through the wide variety of synchronization errors that can happen. Of course, this course is on C++ and modern C++. So, I cannot really teach you on each one of these cases, why it happens, and so on, you will have to read it up elsewhere.

But what these, the features that have already discussed mutex, lock, future and promise, atomic, async one or more of these can be used to solve each one of these situations like conflicting access to a shared memory, shared memory is there and one is accessing that, the other thread is accessing that. And, before one thread has completed the access and put it into a valid state, it is time gets over and the second thread tries to access into gets an invalid value for the shared variable and so on conflicting access, race condition you have already seen.

Deadlock is very simple, that if we are there are two or more resources that two or more threads need to access at the same time, then it is quite possible that one thread has locked one resource and the other thread has locked the other resource. So, none of the threads get the second resource to actually proceed into the critical region. And therefore both of them keep infinitely waiting.

Starvation is a, is a related problem that it may be, as I said that when you unlock which thread gets it next is indeterminable. Like some one of the waiting threads will get it. Now in our example, since we had a given number of threads, and one single iteration, eventually in

whatever order they get, each one of them will get the lock at some point of time and we would be able to complete.

But in a general situation where threads may be getting generated that may be, they may be locking new and new, making new and new locks or trying to make new and new locks. So, when a thread unlocks, it is indeterminable as to which of the waiting threads will get that. So, it is possible that some thread has been waiting waiting waiting several times the opportunity has come for that thread to get the lock but it has eventually not got the lock. So, that is starvation. There maybe priority inversion between higher priority threads and lower priority threads. Lower priority thread getting the lock before the high priority threads and so on.

(Refer Slide Time: 24:31)

The screenshot shows a presentation slide with a blue header and a white content area. The header contains the title 'Synchronization' and a small video feed of the presenter. The content area lists several bullet points:

- A program may need multiple threads to share some data
- If access is not controlled to be sequential, then shared data may become corrupted
 - One thread accesses the data, begins to modify the data, and then is put to sleep because its time slice has expired. The problem arises when the data is in an incomplete state of modification.
 - Another thread awakes and accesses the data, that is only partially modified. The result is very likely to be corrupt data.
- The process of making access serial is called serialization or synchronization
- Synchronization may be achieved in various ways including:
 - *Mutex* (self-study)
 - *Lock* (self-study)
 - *Atomic* (self-study)
 - *Condition Variable* (self-study)
 - *Future and Promises* (self-study)
 - *Async* (self-study)

The slide also features a navigation menu on the left side with items like 'Module M59', 'Partha Pratim Das', 'Objectives & Outlines', 'Threads', 'Races', 'Solutions by Mutex', 'Solutions by Lock', 'Solutions by Atomic', 'Solutions by Future', 'Solutions by Async', 'Synchronization', 'Thread Local', 'Self-Study', 'Minimal Extension', 'Lock', 'Deadlock', 'Atomic', 'Condition Variables', 'Future and Promises', 'Async', 'Practice Examples', and 'Module Summary'. At the bottom, it says 'Programming in Modern C++', 'Partha Pratim Das', and 'M59.28'.

So, variety of such synchronization errors are possible. And it is, this happens because in a in a multi threaded program you need to share data. So, certain accesses which are need to be which need to be controlled to be sequential must be done so through synchronization. This process of making access serial is called serialization or synchronization. And there are several ways of doing that.

And we have except for condition variable which is particularly used for deadlock prevention, all other five we have seen examples of. And at the end I have given self-study module for you to go through each one of them and with sample programs attached with them.

(Refer Slide Time: 25:25)

Synchronization: thread_local

Module M59
Partha Pratim Das

Objectives & Outlines
Threads
Races
Solution by Mutex
Solution by Lock
Solution by Atomic
Solution by Future
Solution by Async

Synchronization
Thread Local
Self-Study
Mutual Exclusion
Locks
Readlock
Atomic
Condition Variables
Futures and Promises
Async
Practical Examples
Module Summary

Synchronization: thread_local

Sources:

- Thread-local storage, isocpp.org
- Storage class specifiers, cppreference
- Thread-Local Data, modernesccpp, 2016
- What does the thread_local mean in C++11?, stackoverflow

Programming in Modern C++ Partha Pratim Das M59.29

Another very interesting concept that, that exists in terms of multi threading is the concept of a thread specific lifetime.

(Refer Slide Time: 25:36)

thread_local

Module M59
Partha Pratim Das

Objectives & Outlines
Threads
Races
Solution by Mutex
Solution by Lock
Solution by Atomic
Solution by Async

Synchronization
Thread Local
Self-Study
Mutual Exclusion
Locks
Readlock
Atomic
Condition Variables
Futures and Promises
Async
Practical Examples
Module Summary

- **thread_local** is a storage class specifier. Thread local data will be created for each thread as needed
- **thread_local** data exclusively belongs to the thread and behaves like **static** data
- Created at its first use and *lifetime* bound to the *lifetime of the thread* (lifetime in **Module 13, 23, & 35**)

```
Thread Local
#include <iostream>
#include <thread>
#include <vector>
using namespace std;
thread_local int i = 0; // Tn1

void f(int newval) { i = newval; }
void g() { cout << i; }
void threadfunc(int id) {
    f(id); ++i; /* */ g();
}

int main() { i = 9;
    vector<thread> th;
    for(int i = 1; i < 4; ++i)
        th.push_back(thread(threadfunc, i));
    for(auto& t: th) t.join();
    cout << i << endl;
}
// 2349, 3249, 4239, 4329, 2439 or 3429

Global
#include <iostream>
#include <thread>
#include <vector>
using namespace std;
int i = 0;

void f(int newval) { i = newval; }
void g() { cout << i; }
void threadfunc(int id) {
    f(id); ++i; /* */ g();
}

int main() { i = 9;
    vector<thread> th;
    for(int i = 1; i < 4; ++i)
        th.push_back(thread(threadfunc, i));
    for(auto& t: th) t.join();
    cout << i << endl;
}
```

Programming in Modern C++ Partha Pratim Das M59.30

We have talked about object lifetime, automatic objects, which are on the stack based on the lexical scope, they have lifetime. We have static objects, which have lifetime which is much wider. We have global statics, which will have a lifetime from the start of main to after the end of main. We may have local statics in namespace or in function scope, which has

a lifetime from the point of creation to the end of the I mean beyond the end of the program and so on.

We have dynamic lifetime we choose our managers. Similarly, we add another lifetime here, which is called `thread_local`. The thread is created and an execution is happening. So, naturally, if a trade is created, an execution is happening, the thread has its own stack has to have otherwise functions could not have been called. With one stack, you cannot have two threads running. So, each thread has a stack of its own.

So, automatic will get managed there, the statics will get managed in a global context, because it is common for all threads. But if I want that something which is, which has a lifetime as that on the thread, but on access, which is as that of the global. So, it is like this, we can define a for example, here ignore this comma. Suppose, if I have a global variable `i`, then this function `f` and function `g` can communicate and function `thread` function can communicate by updating this `i`.

For example, this updates `i`, this writes to `i`, this reads `i` and so on. And this is being done using it as a global. Now, if I do that, then any other thread which may be working with the same thread function, its own it will also get affected because it is a global. So, that solution is to have `thread local` before this. What it means that if it is `thread local`, then as long as you are in one thread, this will look to you as if it is a global.

But if I have two threads, `thread 1` and `thread 2`, then I have two instances of `i`, two different ones. So, that the functions in `thread 1` will share the `thread_local` of `thread 1`. The functions of `thread 2` will share the `thread_local` of `thread 2`. So, that gives a nice advantage in terms of programming in some cases.

(Refer Slide Time: 28:58)

The screenshot shows a presentation slide with a blue header and a white main area. The header contains the text 'Self-Study' and a small video feed of a man in the top right corner. The left sidebar is a dark blue vertical menu with white text listing various topics: 'Module M59', 'Partha Pratim Das', 'Objectives & Outlines', 'Threads', 'Races', 'Solution by Mutex', 'Solution by Lock', 'Solution by Atomic', 'Solution by Future', 'Solution by Async', 'Synchronization', 'Thread Local', 'Self-Study', 'Mutual Exclusion', 'Locks', 'Deadlock', 'Atomic', 'Condition Variables', 'Future and Promise', 'Async', 'Practice Examples', and 'Module Summary'. The main white area has the text 'Self-Study' in red. At the bottom, there is a footer with 'Programming in Modern C++', 'Partha Pratim Das', and 'M59.31'.

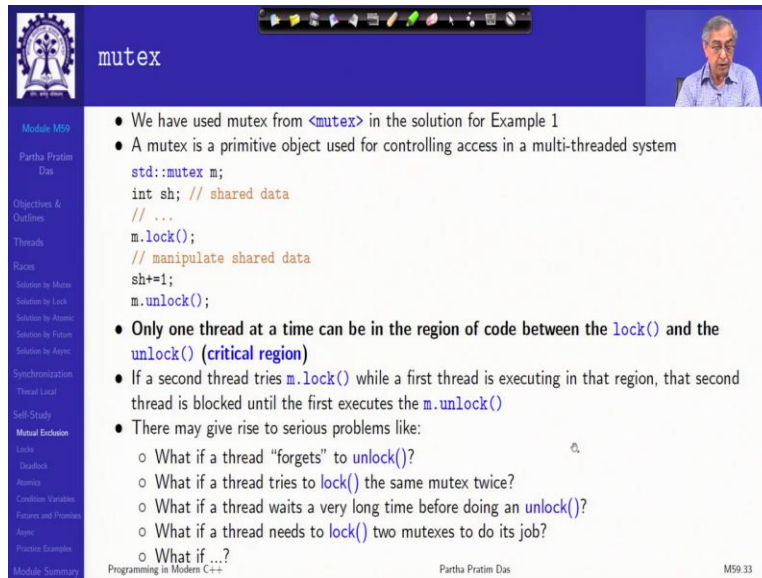
So, this is the short overview that I had for you.

(Refer Slide Time: 29:07)

The screenshot shows a presentation slide with a blue header and a white main area. The header contains the text 'Synchronization: mutex' and a small video feed of a man in the top right corner. The left sidebar is a dark blue vertical menu with white text listing various topics: 'Module M59', 'Partha Pratim Das', 'Objectives & Outlines', 'Threads', 'Races', 'Solution by Mutex', 'Solution by Lock', 'Solution by Atomic', 'Solution by Future', 'Solution by Async', 'Synchronization', 'Thread Local', 'Self-Study', 'Mutual Exclusion', 'Locks', 'Deadlock', 'Atomic', 'Condition Variables', 'Future and Promise', 'Async', 'Practice Examples', and 'Module Summary'. The main white area has the text 'Synchronization: mutex' in red. Below this text, there is a section titled 'Sources:' followed by a bulleted list: '● Mutual exclusion, isocpp.org', '● std::mutex, cplusplus', and '● An Overview of the New C++ (C++11/14), Scott Meyers Training Courses'. At the bottom, there is a footer with 'Programming in Modern C++', 'Partha Pratim Das', and 'M59.32'.

What I have now given is different slides I mean different sections for your self study.

(Refer Slide Time: 29:13)



mutex

- We have used mutex from `<mutex>` in the solution for Example 1
- A mutex is a primitive object used for controlling access in a multi-threaded system

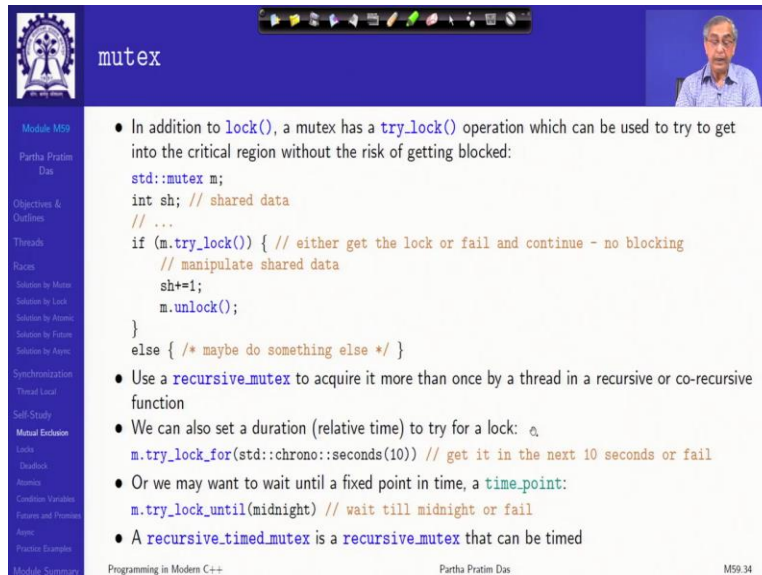
```
std::mutex m;  
int sh; // shared data  
// ...  
m.lock();  
// manipulate shared data  
sh+=1;  
m.unlock();
```

- **Only one thread at a time can be in the region of code between the `lock()` and the `unlock()` (critical region)**
- If a second thread tries `m.lock()` while a first thread is executing in that region, that second thread is blocked until the first executes the `m.unlock()`
- There may give rise to serious problems like:
 - What if a thread “forgets” to `unlock()`?
 - What if a thread tries to `lock()` the same mutex twice?
 - What if a thread waits a very long time before doing an `unlock()`?
 - What if a thread needs to `lock()` two mutexes to do its job?
 - What if ...?

Programming in Modern C++ Partha Pratim Das M59.33

This is on mutex, which talks a little bit more about mutex.

(Refer Slide Time: 29:16)



mutex

- In addition to `lock()`, a mutex has a `try_lock()` operation which can be used to try to get into the critical region without the risk of getting blocked:

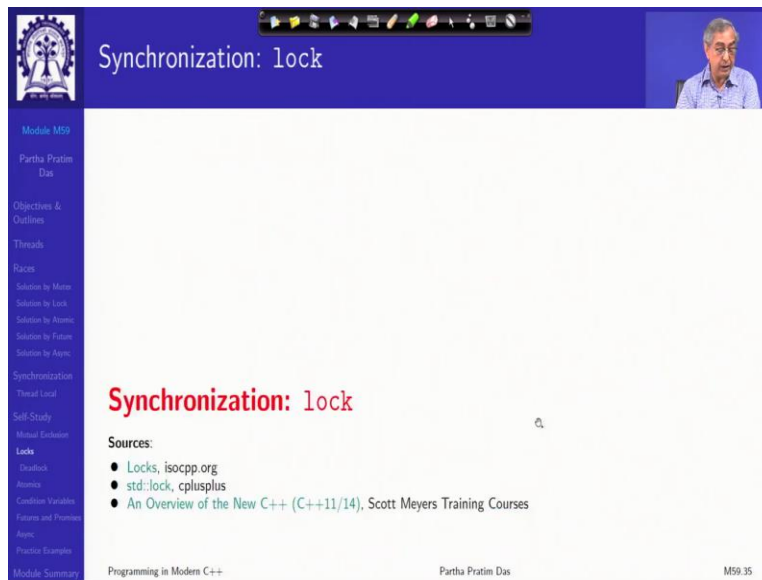
```
std::mutex m;  
int sh; // shared data  
// ...  
if (m.try_lock()) { // either get the lock or fail and continue - no blocking  
    // manipulate shared data  
    sh+=1;  
    m.unlock();  
}  
else { /* maybe do something else */ }
```

- Use a `recursive_mutex` to acquire it more than once by a thread in a recursive or co-recursive function
- We can also set a duration (relative time) to try for a lock: `m.try_lock_for(std::chrono::seconds(10))` // get it in the next 10 seconds or fail
- Or we may want to wait until a fixed point in time, a `time_point`: `m.try_lock_until(midnight)` // wait till midnight or fail
- A `recursive_timed_mutex` is a `recursive_mutex` that can be timed

Programming in Modern C++ Partha Pratim Das M59.34

You have already seen the example use. And the specific features like besides lock and unlock it as a `try_lock`. So, learn what it is.

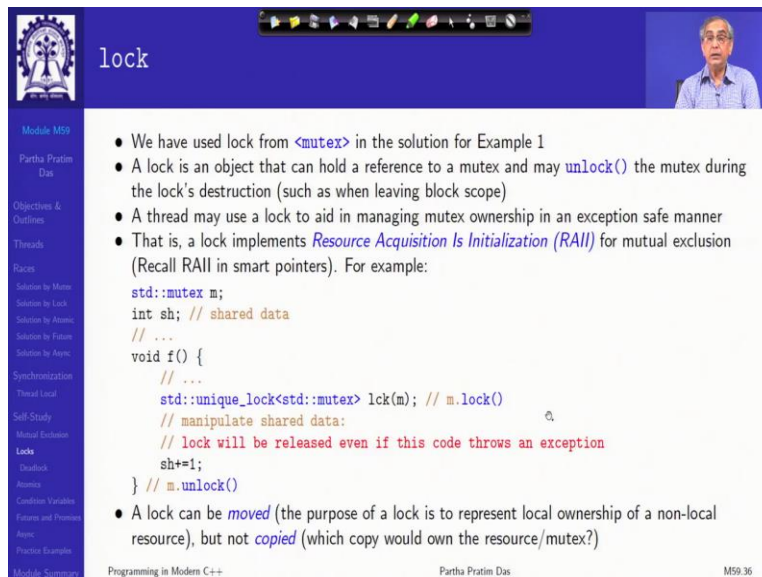
(Refer Slide Time: 29:24)



The slide is titled "Synchronization: Lock". On the left, there is a navigation menu for "Module M59" by Partha Pratim Das, listing various topics like "Objective & Outlines", "Threads", "Races", "Synchronization", "Thread Local", "Self-Study", "Mutual Exclusion", "Locks", "Deadlock", "Atomic", "Condition Variable", "Future and Promise", "Async", "Practice Examples", and "Module Summary". The main content area has the title "Synchronization: lock" in red. Below it, under "Sources:", there is a list of three items: "Locks, isocpp.org", "std::lock, cplusplus", and "An Overview of the New C++ (C++11/14), Scott Meyers Training Courses". The footer contains "Programming in Modern C++", "Partha Pratim Das", and "M59.35".

Then I have given more slides on the lock itself.

(Refer Slide Time: 29:27)



The slide is titled "lock". It contains a list of bullet points explaining the concept of a lock and its use in C++:

- We have used lock from `<mutex>` in the solution for Example 1
- A lock is an object that can hold a reference to a mutex and may `unlock()` the mutex during the lock's destruction (such as when leaving block scope)
- A thread may use a lock to aid in managing mutex ownership in an exception safe manner
- That is, a lock implements *Resource Acquisition Is Initialization (RAII)* for mutual exclusion (Recall RAII in smart pointers). For example:

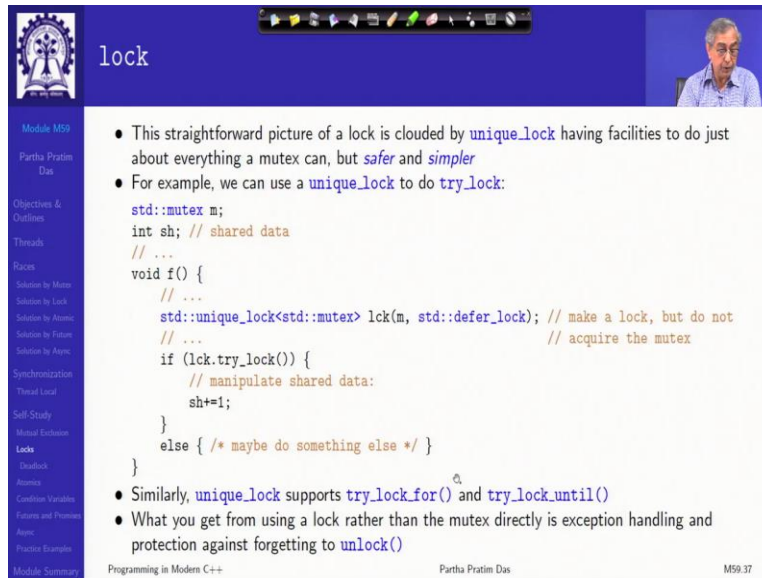
```
std::mutex m;
int sh; // shared data
// ...
void f() {
    // ...
    std::unique_lock<std::mutex> lck(m); // m.lock()
    // manipulate shared data:
    // lock will be released even if this code throws an exception
    sh+=1;
} // m.unlock()
```

- A lock can be *moved* (the purpose of a lock is to represent local ownership of a non-local resource), but not *copied* (which copy would own the resource/mutex?)

The footer contains "Programming in Modern C++", "Partha Pratim Das", and "M59.36".

Which we have seen already how what that locks can be moved but they cannot be copied and so on so forth.

(Refer Slide Time: 29:38)



lock

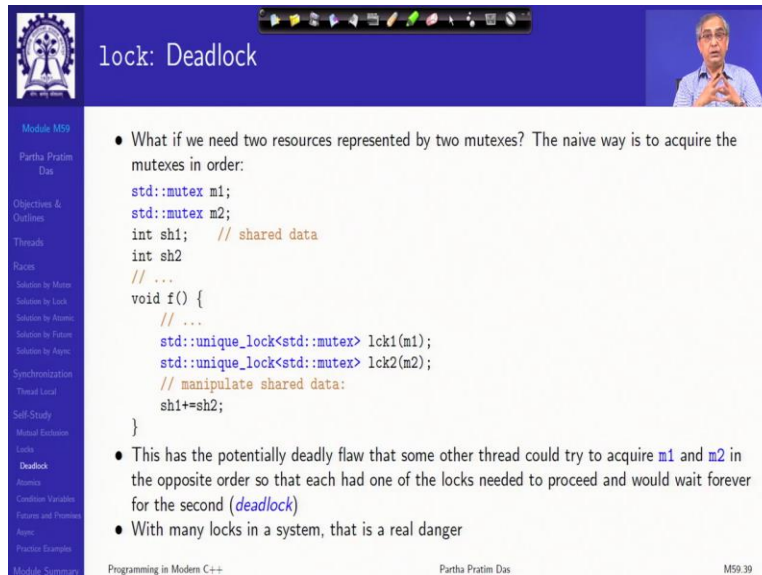
- This straightforward picture of a lock is clouded by `unique_lock` having facilities to do just about everything a mutex can, but *safer* and *simpler*
- For example, we can use a `unique_lock` to do `try_lock`:

```
std::mutex m;
int sh; // shared data
// ...
void f() {
    // ...
    std::unique_lock<std::mutex> lck(m, std::defer_lock); // make a lock, but do not
    // ... // acquire the mutex
    if (lck.try_lock()) {
        // manipulate shared data:
        sh+=1;
    }
    else { /* maybe do something else */ }
}
```
- Similarly, `unique_lock` supports `try_lock_for()` and `try_lock_until()`
- What you get from using a lock rather than the mutex directly is exception handling and protection against forgetting to `unlock()`

Programming in Modern C++ Partha Pratim Das M59.37

Very specifically, how locks can be used.

(Refer Slide Time: 29:43)



lock: Deadlock

- What if we need two resources represented by two mutexes? The naive way is to acquire the mutexes in order:

```
std::mutex m1;
std::mutex m2;
int sh1; // shared data
int sh2
// ...
void f() {
    // ...
    std::unique_lock<std::mutex> lck1(m1);
    std::unique_lock<std::mutex> lck2(m2);
    // manipulate shared data:
    sh1+=sh2;
}
```
- This has the potentially deadly flaw that some other thread could try to acquire `m1` and `m2` in the opposite order so that each had one of the locks needed to proceed and would wait forever for the second (*deadlock*)
- With many locks in a system, that is a real danger

Programming in Modern C++ Partha Pratim Das M59.39

I mean what is the C++11 support mechanism to use locks in a way so that inherently deadlock can be prevented.

(Refer Slide Time: 29:53)

atomic

- We have used atomic from `<atomic>` in the solution for Example 1
- Each instantiation of the `std::atomic` template defines an atomic type. **If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined**
- `std::atomic` is neither copyable nor movable
- Type aliases are provided for `bool` (`std::atomic_bool`) and integral types like `int`, `short`, etc.

```
#include <iostream> // std::cout
#include <atomic> // std::atomic, std::atomic_flag, ATOMIC_FLAG_INIT
#include <thread> // std::thread, std::this_thread::yield
#include <vector> // std::vector
std::atomic<bool> ready (false);
std::atomic_flag winner = ATOMIC_FLAG_INIT; // set false. atomic_flag has no load / store
void countm (int id) {
    while (!ready) { std::this_thread::yield(); } // all threads wait for the ready signal to start
    for (volatile int i=0; i<1000000; ++i) // go!, count to 1 million
        if (!winner.test_and_set()) // atomically sets the flag to true and obtains its previous value
            { std::cout << "thread #" << id << " won!\n"; }
};
int main () { std::vector<std::thread> threads;
    std::cout << "spawning 10 threads that count to 1 million...\n";
    for (int i=1; i<=10; ++i) threads.push_back(std::thread(countm,i));
    ready = true; // signal ready to start
    for (auto& th : threads) th.join();
} // thread #8 won! // thread #4 won! // thread #1 won! // thread #9 won! ...
```

Programming in Modern C++ Partha Pratim Das M59.42

Then more on the atomic variables which I have given other examples condition variables, futures, async each one of them there are more.

(Refer Slide Time: 30:16)

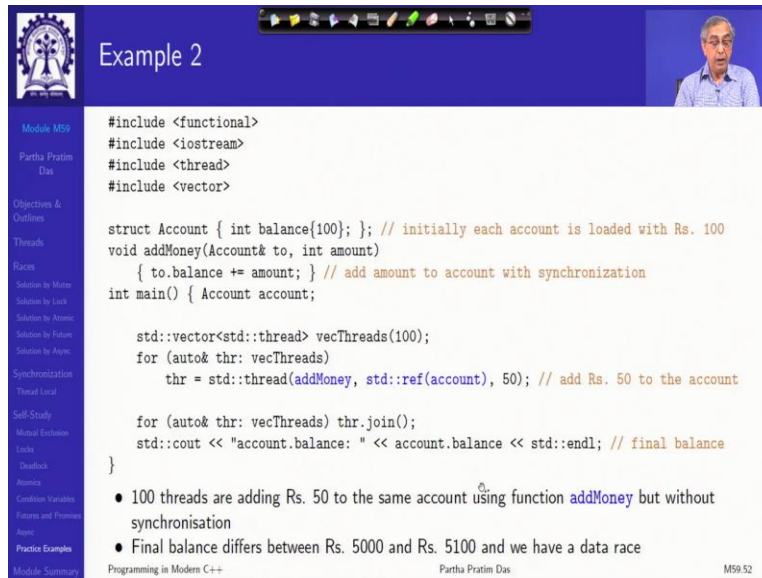
Race Condition and Data Race: Example 2

Race Condition and Data Race: Practice Examples

Programming in Modern C++ Partha Pratim Das M59.51

And then we have two more examples.

(Refer Slide Time: 30:09)



Example 2

```
#include <functional>
#include <iostream>
#include <thread>
#include <vector>

struct Account { int balance{100}; }; // initially each account is loaded with Rs. 100
void addMoney(Account& to, int amount)
{ to.balance += amount; } // add amount to account with synchronization
int main() { Account account;

    std::vector<std::thread> vecThreads(100);
    for (auto& thr: vecThreads)
        thr = std::thread(addMoney, std::ref(account), 50); // add Rs. 50 to the account

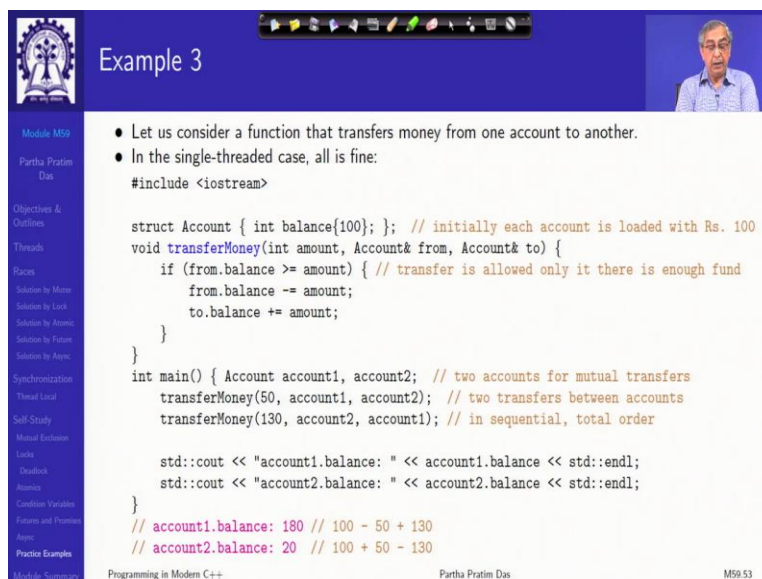
    for (auto& thr: vecThreads) thr.join();
    std::cout << "account.balance: " << account.balance << std::endl; // final balance
}
```

- 100 threads are adding Rs. 50 to the same account using function `addMoney` but without synchronisation
- Final balance differs between Rs. 5000 and Rs. 5100 and we have a data race

Programming in Modern C++ Partha Pratim Das M59.52

One in which, there is an account to which multiple threads are adding an amount. They are the same synchronization problem will arise. So, because multiple threads are, so multiple threads are actually updating their account. So, that is a situation and what you will have to do is to use those mechanisms and convert it into a safe program. Obviously you will have to use the, the random delay and the repeat process to be able to get the right set up.

(Refer Slide Time: 30:46)



Example 3

- Let us consider a function that transfers money from one account to another.
- In the single-threaded case, all is fine:

```
#include <iostream>

struct Account { int balance{100}; }; // initially each account is loaded with Rs. 100
void transferMoney(int amount, Account& from, Account& to) {
    if (from.balance >= amount) { // transfer is allowed only if there is enough fund
        from.balance -= amount;
        to.balance += amount;
    }
}

int main() { Account account1, account2; // two accounts for mutual transfers
    transferMoney(50, account1, account2); // two transfers between accounts
    transferMoney(130, account2, account1); // in sequential, total order

    std::cout << "account1.balance: " << account1.balance << std::endl;
    std::cout << "account2.balance: " << account2.balance << std::endl;
}

// account1.balance: 180 // 100 - 50 + 130
// account2.balance: 20 // 100 + 50 - 130
```

Programming in Modern C++ Partha Pratim Das M59.53

And the other example is where you are transferring money from one account to the other. Now, depending on in which order it happens, your transfer will maybe correct, may not be correct. So, you will have to again make it safe by using a synchronization.

(Refer Slide Time: 31:11)

Module Summary

- Understood synchronization issues in multi-thread programming in C++
- Studied various synchronization mechanisms through example
- Provided detail for self-study of synchronization mechanisms:
 - *Mutex*
 - *Lock*
 - *Atomics*
 - *Condition Variable*
 - *Future and Promises*
 - *Async*
- Explored use of the synchronization mechanisms to alleviate race condition and data race and left practice examples

Module M59
Partha Pratim Das
Objectives & Outlines
Threads
Races
Solution by Mutex
Solution by Lock
Solution by Atomic
Solution by Future
Solution by Async
Synchronization
Thread Local
Self-Study
Mutual Exclusion
Locks
Deadlock
Atomic
Condition Variables
Future and Promises
Async
Practice Examples
Module Summary

Programming in Modern C++ Partha Pratim Das M59/95

So, that was all about concurrencies support in C++11 onwards. And we have studies through the example to keep it manageable. I left a lot of stuff for your self-study to gain further knowledge. Thank you very much, thanks for your attention. And we will meet in the last module.