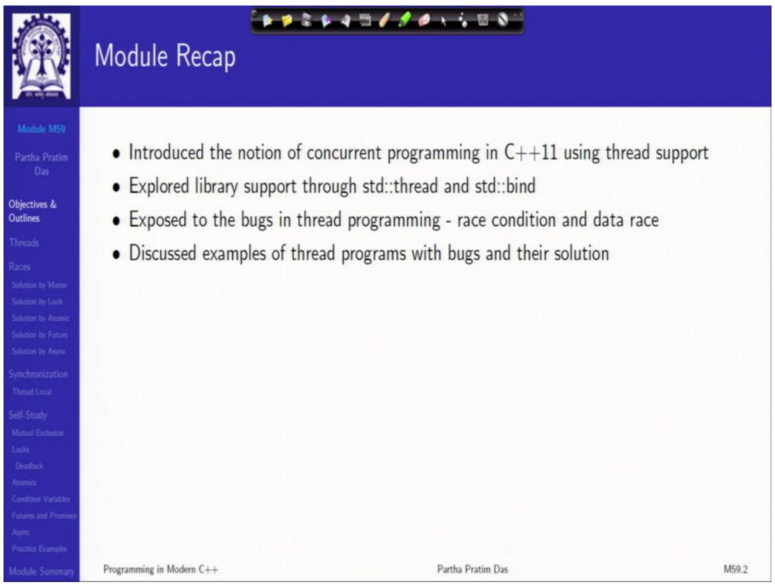
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture – 59
C++ and Beyond: Concurrency: Part 2

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

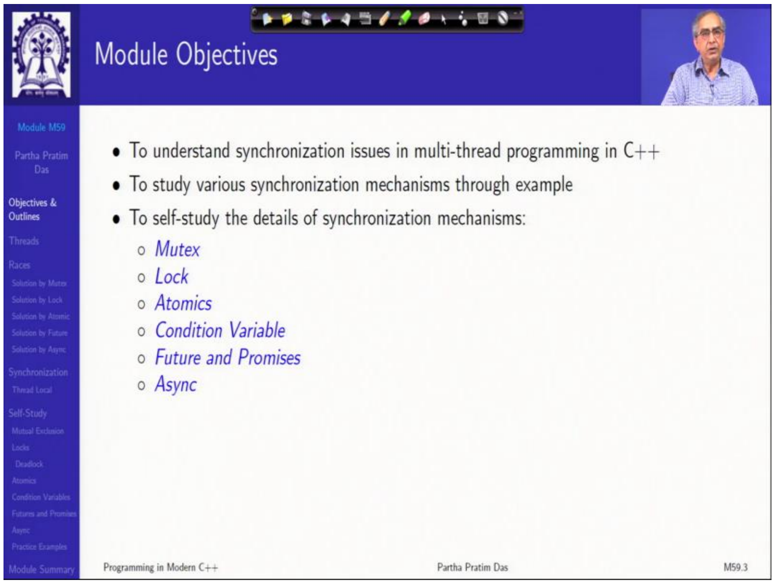
(Refer Slide Time: 00:38)



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.

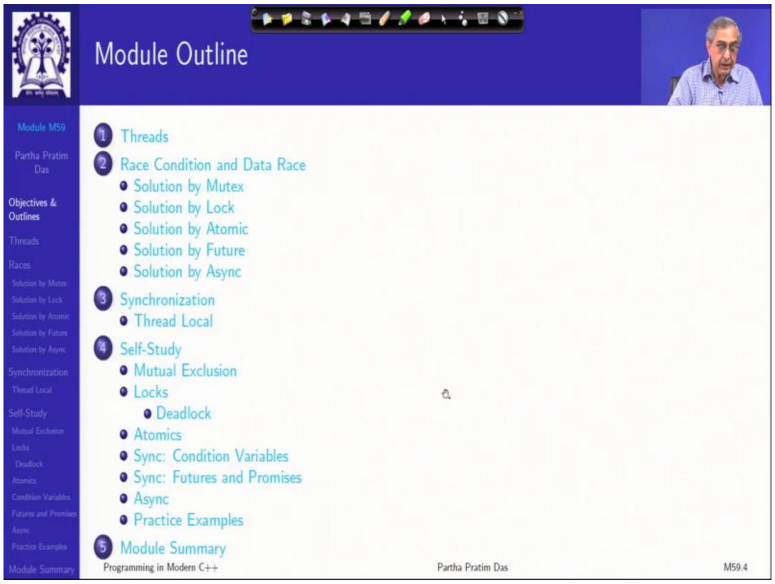
(Refer Slide Time: 01:39)



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.

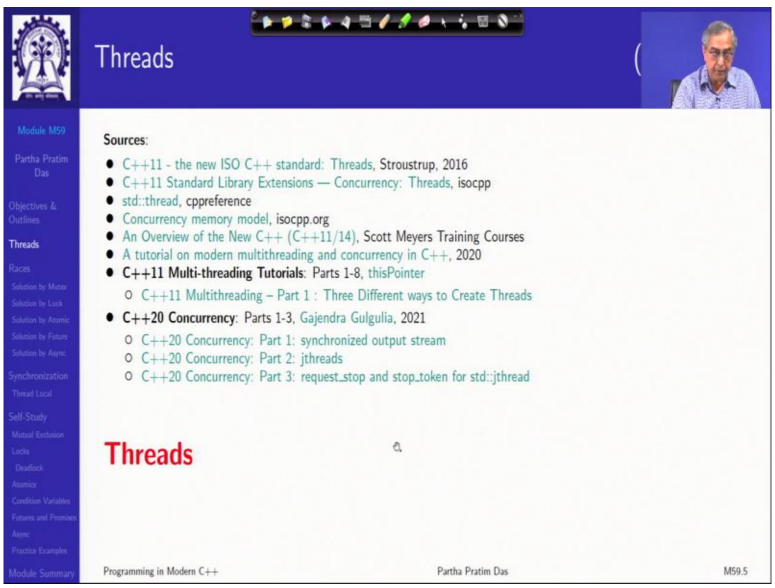
So, what I have done?? I have prepared a set of slides at the end of this presentation under the self-study banner. So, that you can study some details of specific synchronization mechanisms yourself and also practice examples that are given at the end.

(Refer Slide Time: 02:40)



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

(Refer Slide Time: 02:51)

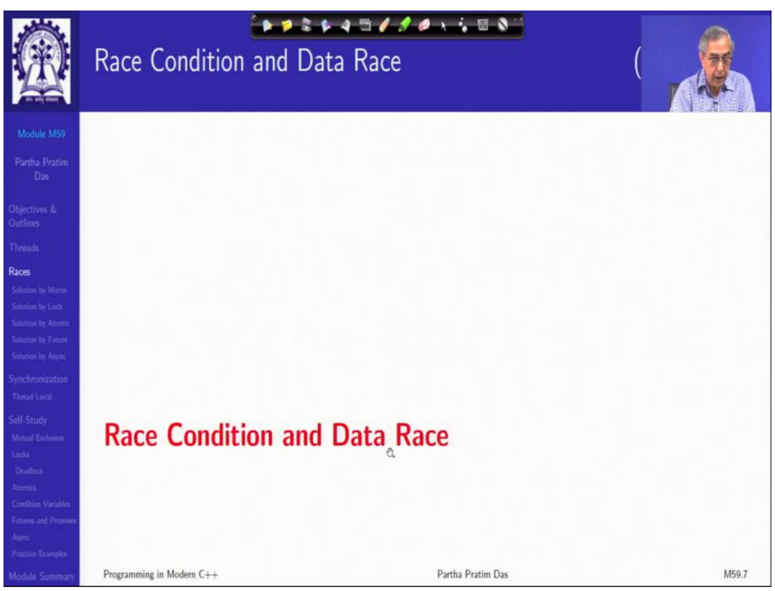


Just a quick recap from the last module.

(Refer Slide Time: 02:52)

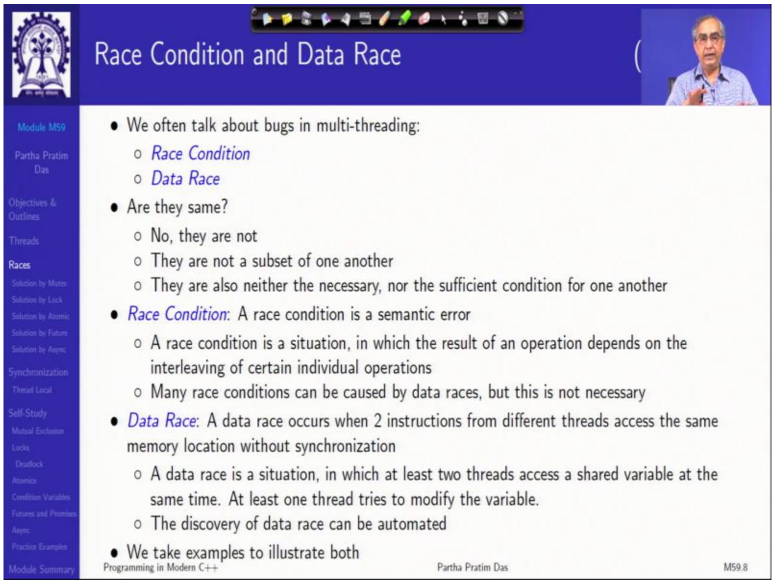
We showed that thread is a lightweight process and we have learned the basic trade operations using the thread component of the standard library, that is creating a thread, passing parameters to a thread directly or by std::bind, returning the result from a thread directly or by std::bind, joining threads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

(Refer Slide Time: 03:23)



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

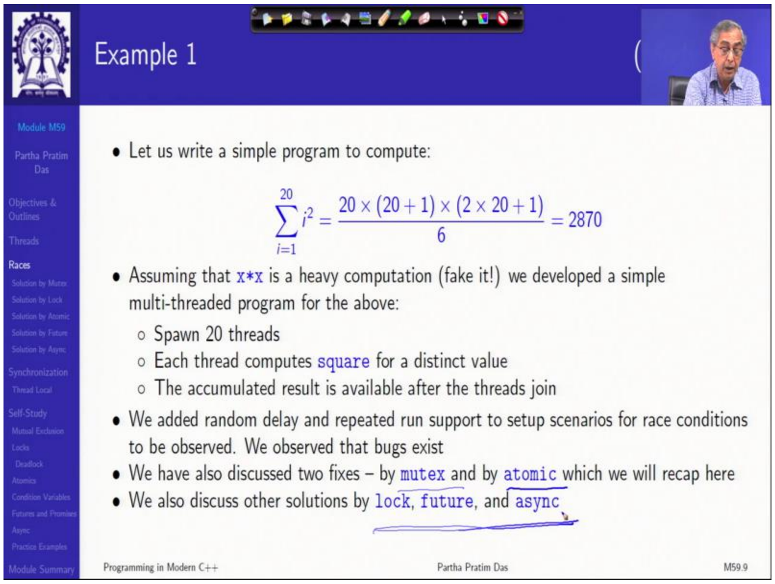
(Refer Slide Time: 03:29)



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)

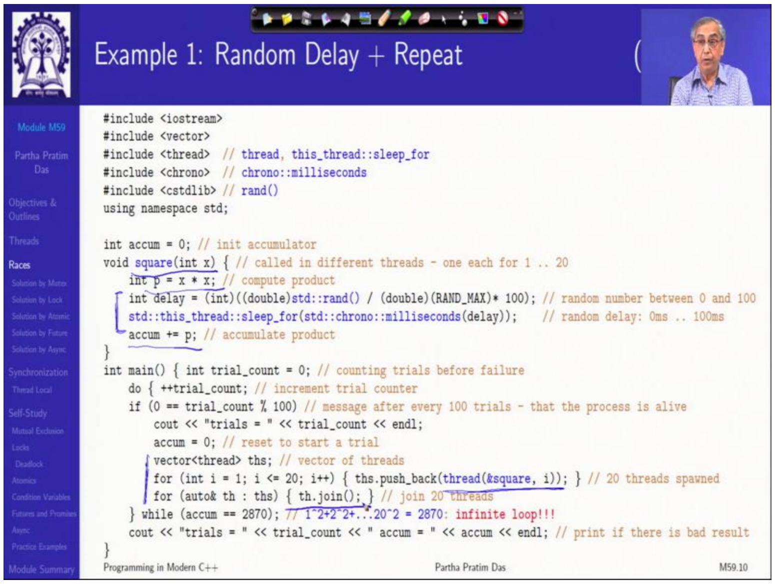


Now, we will continue with the same example which was to find the sum of squares of 1 to 20 and we had assumed that x multiplied by x is a heavy operation to simulate that we have also modeled a random delay after the computation of this product in accumulating the results. And to be able to repeat this, the thread bugs are kind of indeterminate onces. 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.

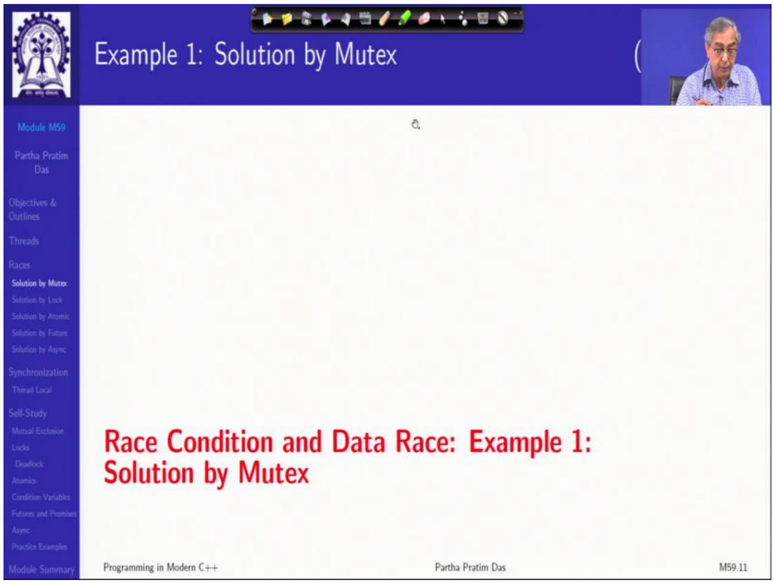
(Refer Slide Time: 05:53)



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.

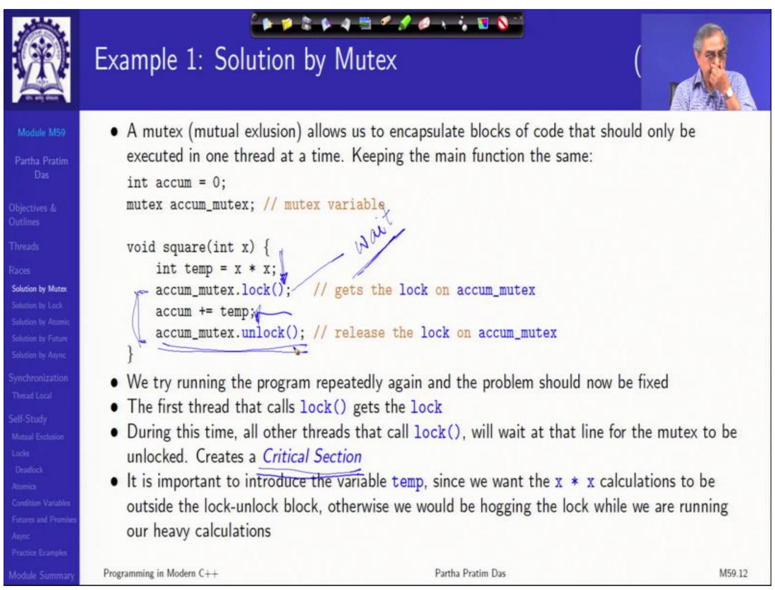
And then put different threads with the associated square function, but different parameters to put them in the vector. So, that as soon as I put them they start working and they will get variable amount of delay amongst them, and then it will accumulate and this accumulation will be completed when all 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.

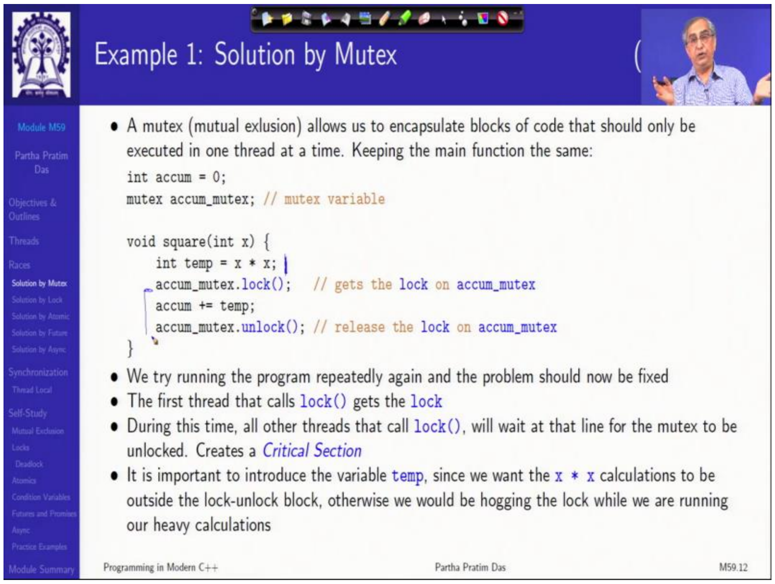
(Refer Slide Time: 07:14)



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

(Refer Slide Time: 07:18)





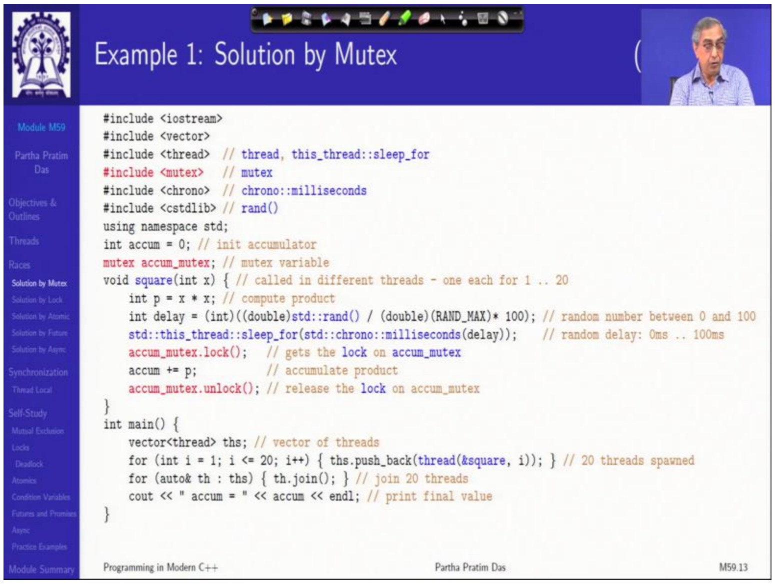
Mutex is like a gate. So, what it creates is what is known as a critical region. So, this is known as a critical section or a critical region. So, what it does is when a thread reaches this point, and assuming that no other thread has reached that point, then it will acquire that lock. So, it can acquire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

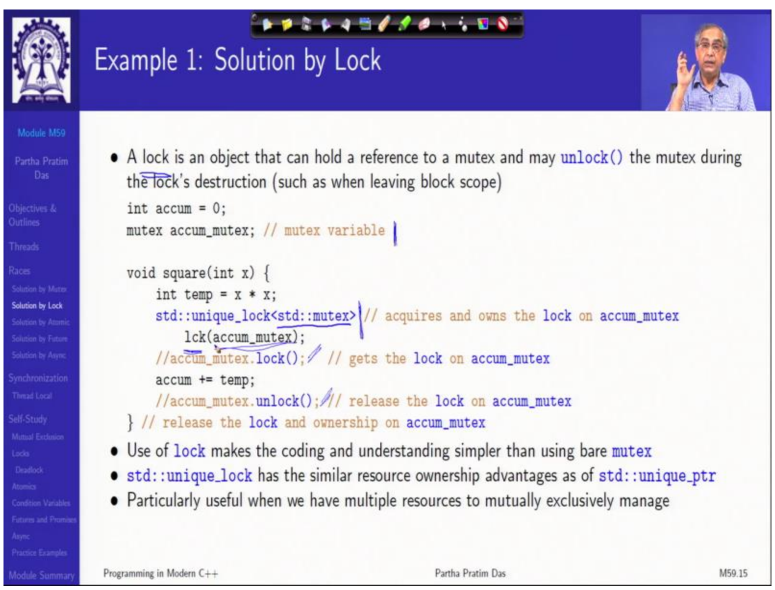
That is at any point of time only one thread will be able to add the product of x into x to the accumulator. So, this will ensure that the problem that we had seen earlier that the old value of an accumulator is being read and being updated will not arise.

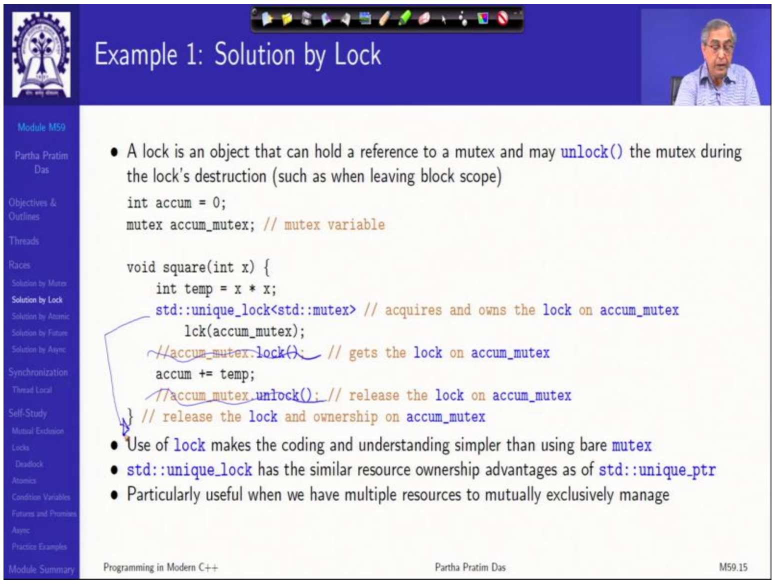
(Refer Slide Time: 09:15)

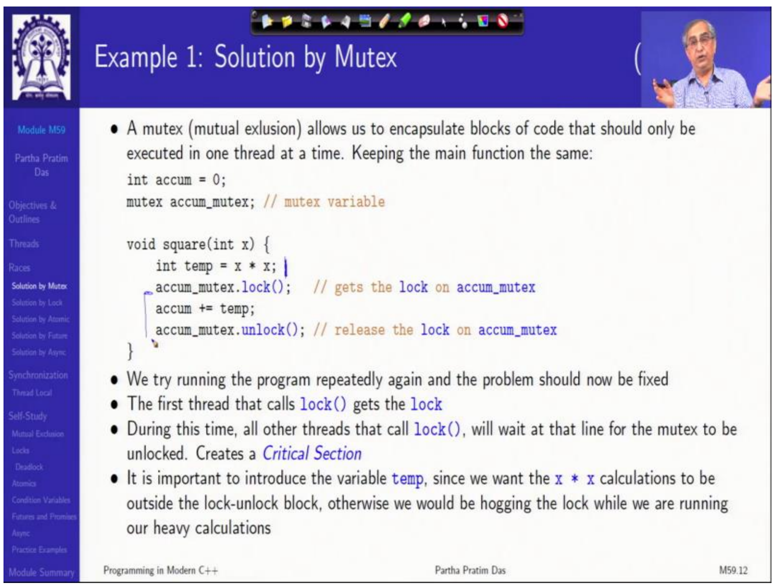


And with that, we have seen the modified solution, the safe solution, whereby red I have shown the changes that you make to the central thread program, of course, I have not shown the reputation part because that's basically an experimental setup, but this is a program which will work correctly.

(Refer Slide Time: 09:34)







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.

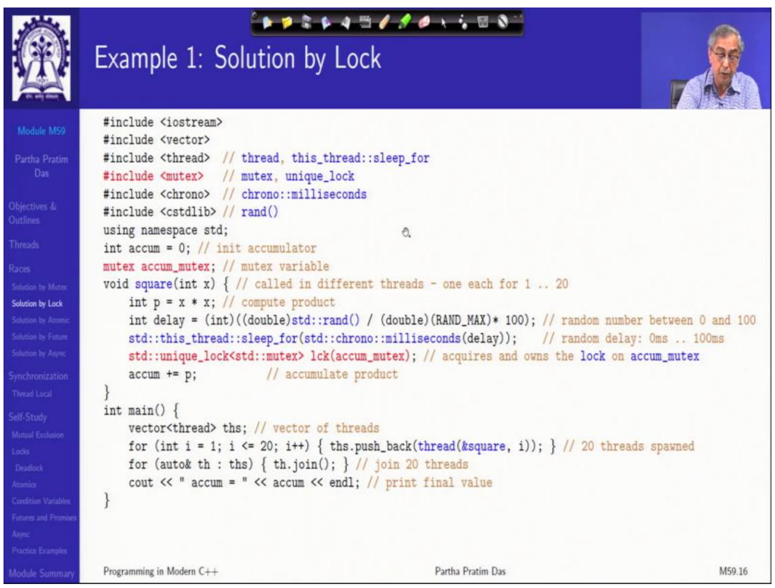
But instead of doing that, what you do is you do this. That is you say std::unique_lock, which is available in the mutex component and pass a template parameter mutex, then this is your lock variable, and this is the mutex that you are locking. So, what you say is basically, a unique_lock is a template 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.

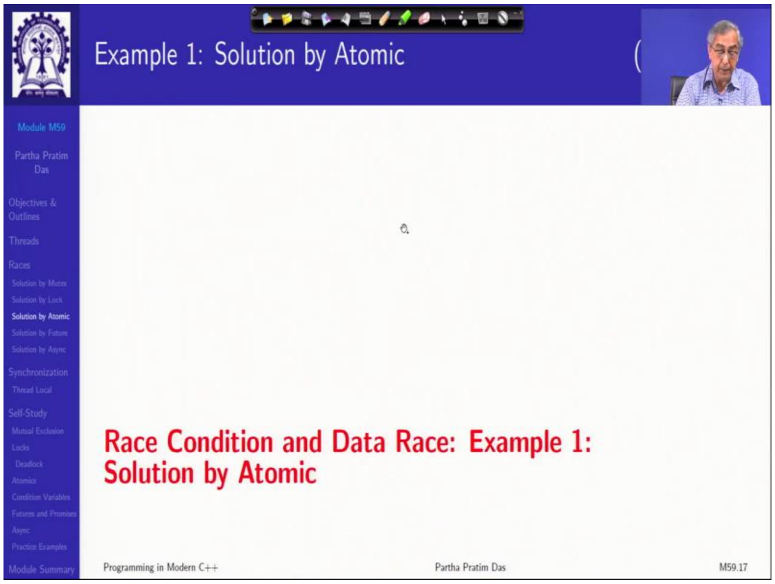
And when the unique pointer that goes out of scope, its destructor automatically destroys the object pointed to by that pointer and disappears. So, exactly in the same way unit lock achieves a resource management for the locks, which makes programming far more easier.

(Refer Slide Time: 12:28)



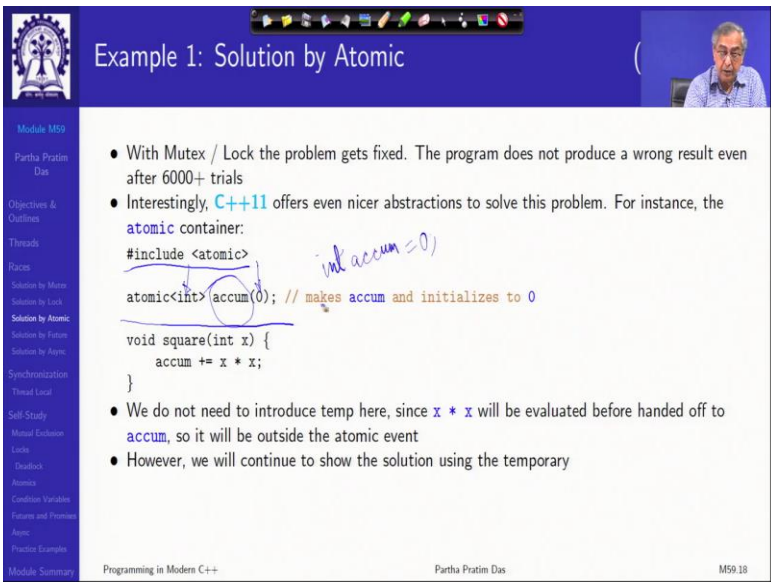
So, using the lock, if we right now, then the program becomes actually much simpler, we will still need to define the mutex variable, but it is just one line of unique lock that before the accumulation of the product which solves the problem. So, this is the second way of doing this.

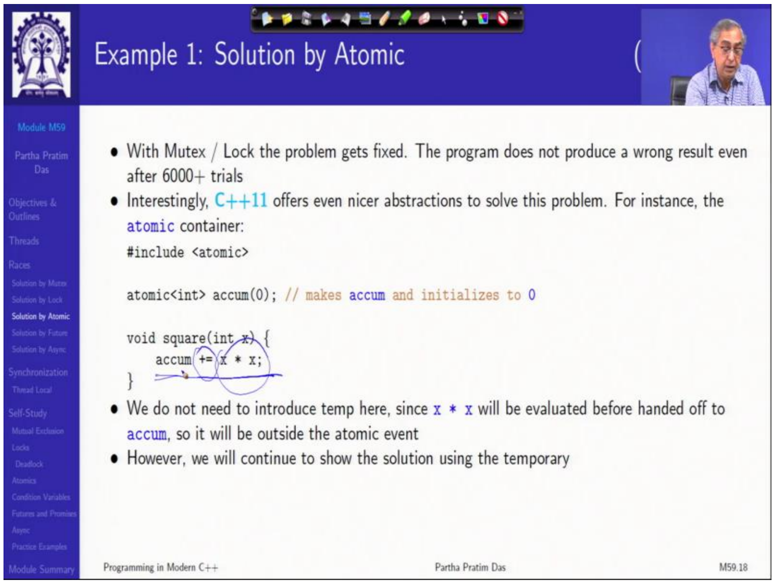
(Refer Slide Time: 12:49)



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

(Refer Slide Time: 12:55)



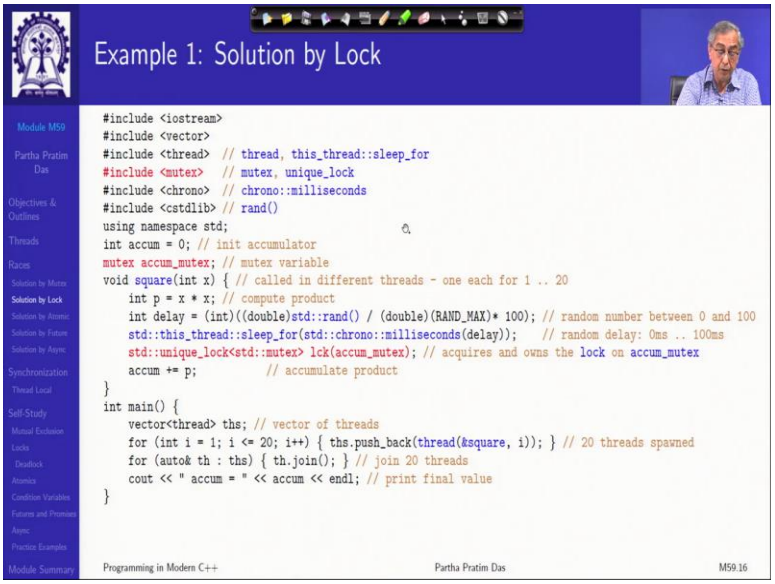


The third solution is what we have seen in the last module also that you have a component atomic by which any variable which needs to be updated in a critical section or needs to be updated safely by multiple threads can be declared as automatic. So, I am, I am reexplaining that so this is a component. 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 acuum, the atomic behavior that is defined in the library will make sure that threads are properly serialized.

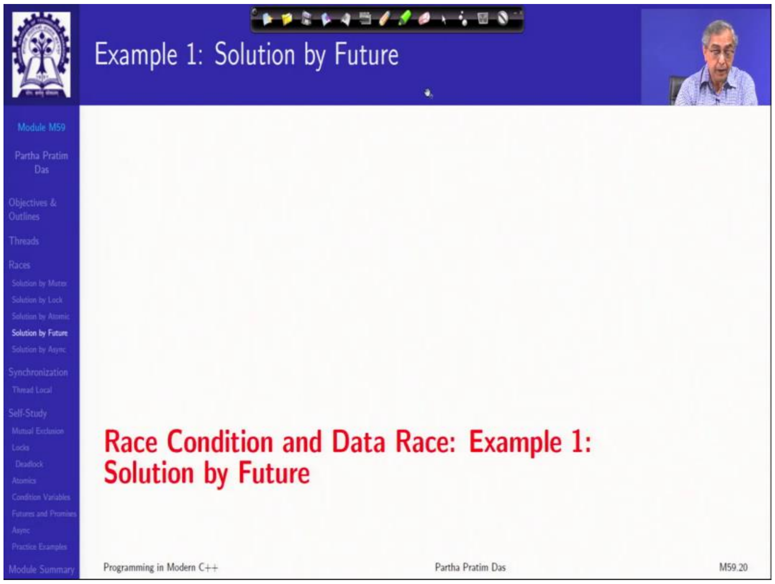
And this update happens as an atomic operation that is when one thread is doing the update that it has read the value of the accumulator and is adding the product to it and getting the final value. During this time, no other thread will be able to do that. So, this atomicity is what is important and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

(Refer Slide Time: 14:36)



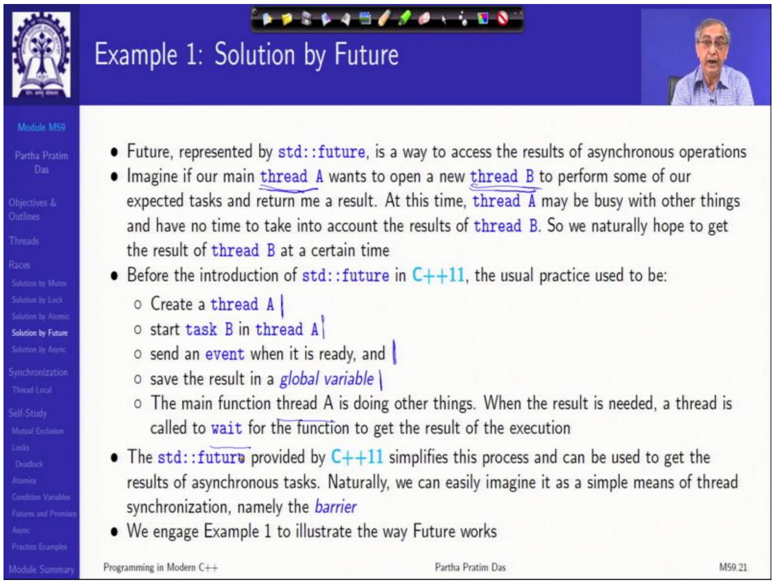
So, again with this, there is almost no change in your earlier unsafe program. All that you need is replaced the global declaration with the declaration of atomic int for accum with initialization 0.

(Refer Slide Time: 14:55)



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

(Refer Slide Time: 15:02)

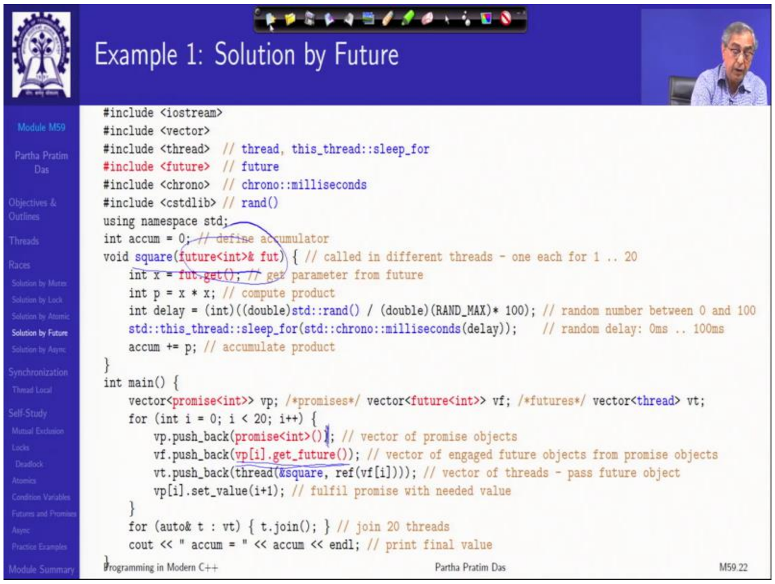


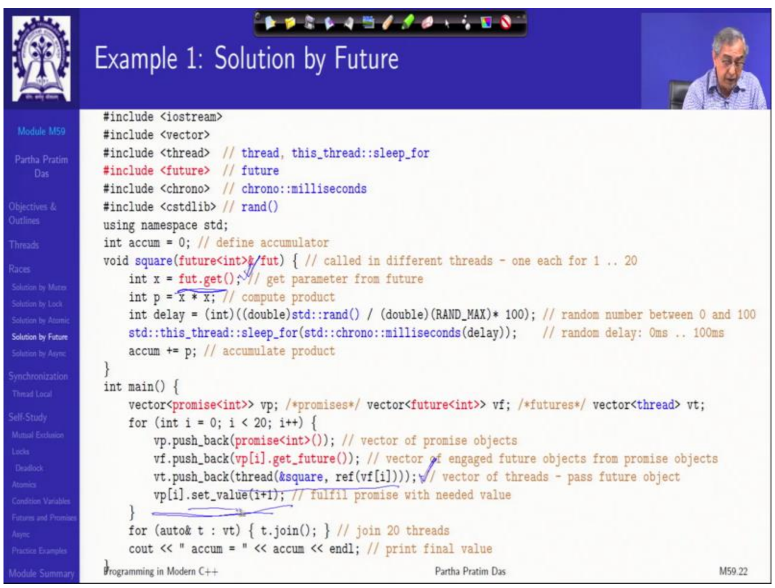
So, future is kind of a way to access the result of an asynchronous operation. You know, threads are doing things on their own. So, there is no as such, they are not synchronized, so they are doing things, someone is computing, someone else need to use that. So, if you have to access the result of an 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)



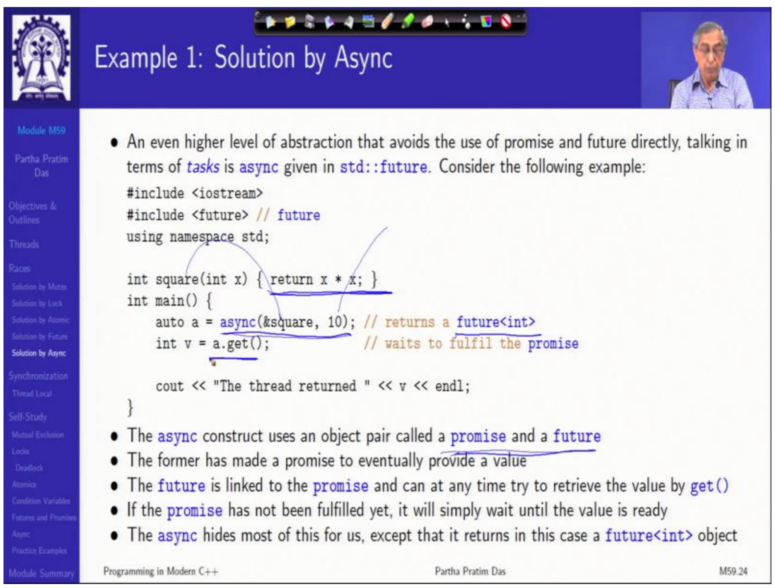


So, what it does, it basically initially creates a promise that I will give you, I will give you this in future. Then from the promise you get a future, you get a handle to a future. And that future is what you pass to your task, you pass that future to the task. And in the task, you can, you can obviously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

Then your task, then you create this threat 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)

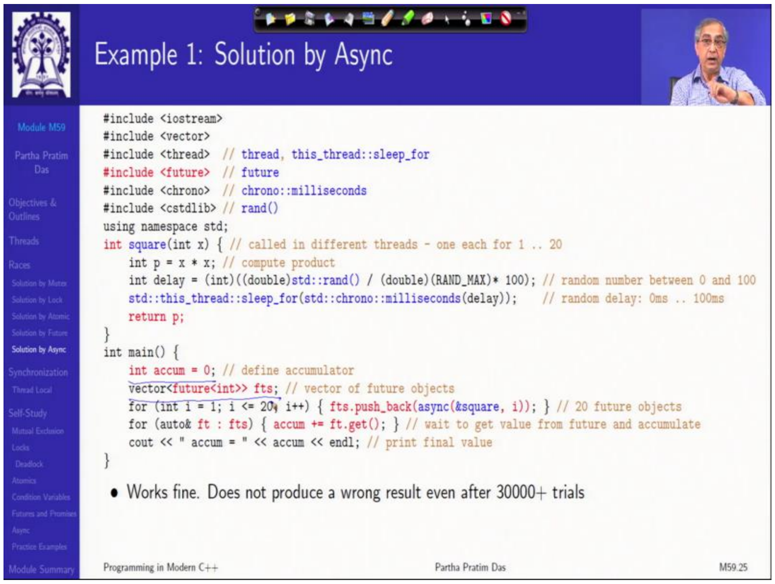


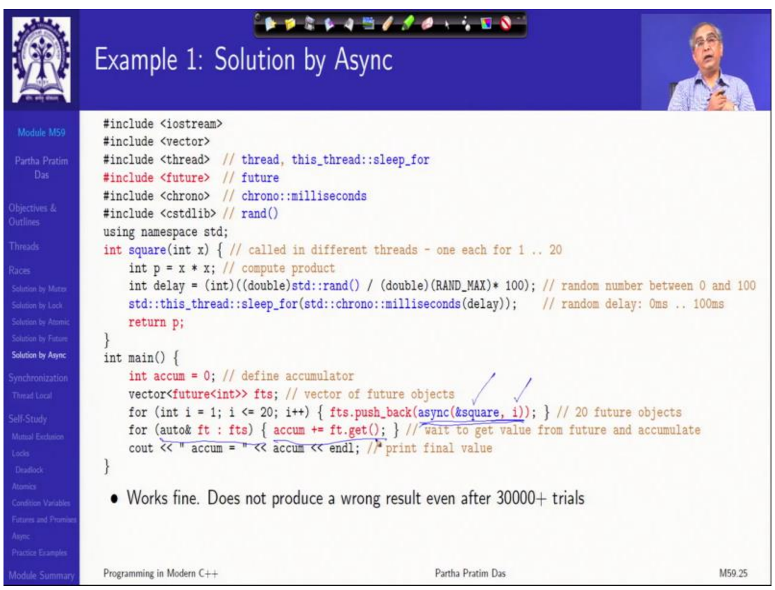
The fifth way to solve this problem is actually to package future and promise into something of a higher abstraction called async. What async does is it you have seen that you need a pair of promise and future to solve the problem. So, the question is if we always need that pair, why not define a high 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)



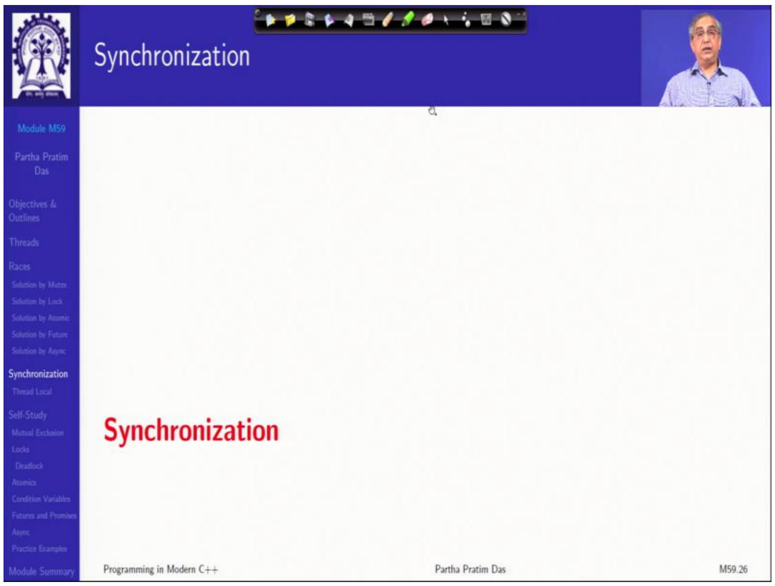


So, if we have to use this, now we have an accumulator again. And we have a, I have put the accumulator as a part of main instead of global, because my task function square, no more uses the accumulator. That task function is not adding it to the accumulator, because that is the synchronizable part. 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.

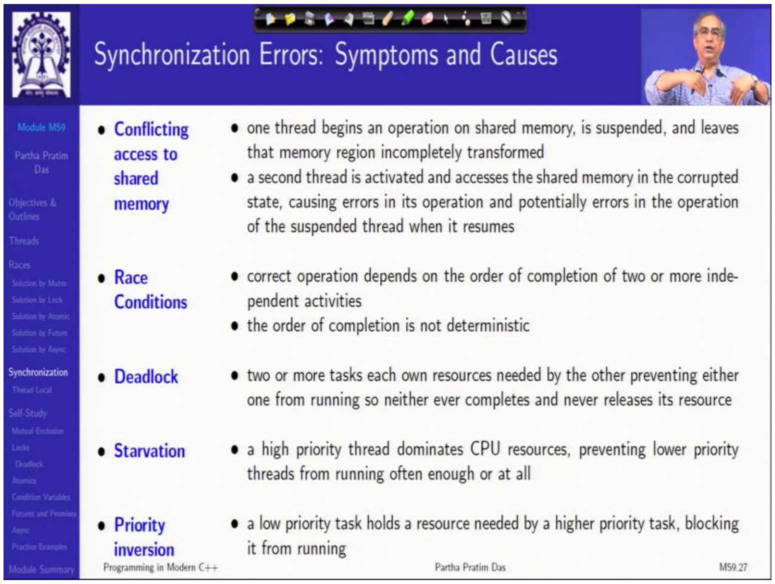
So, I do ft, that that future object, I do a get, and I wait, if it has not finished, and as soon as I get I add it to the accumulator. So, this happens one after the other. So, this part is serialized as I wanted, and I get a direct result.

(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)



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.

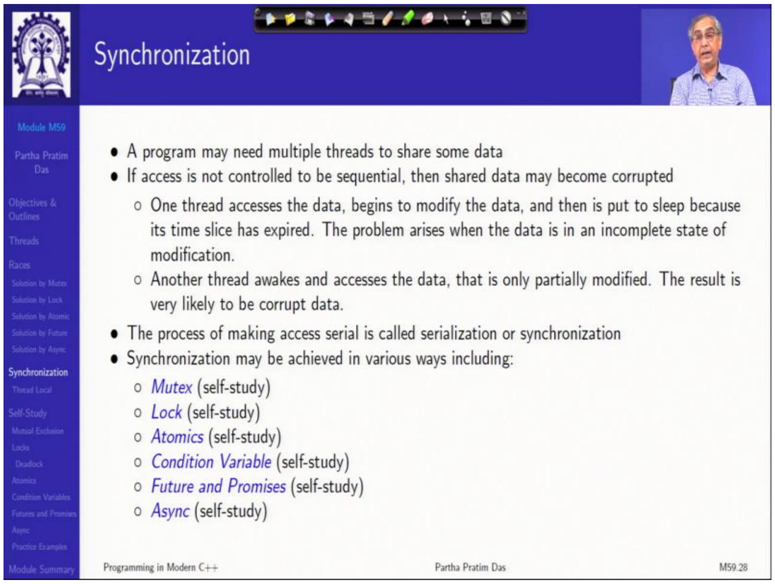
But what these, the features that have already discussed mutex, lock, future and promise, atomic, async one or more of these can be used to solve each one of these situations like conflicting access to a shared memory, shared memory is there and one is accessing that, the other thread is accessing that. 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.

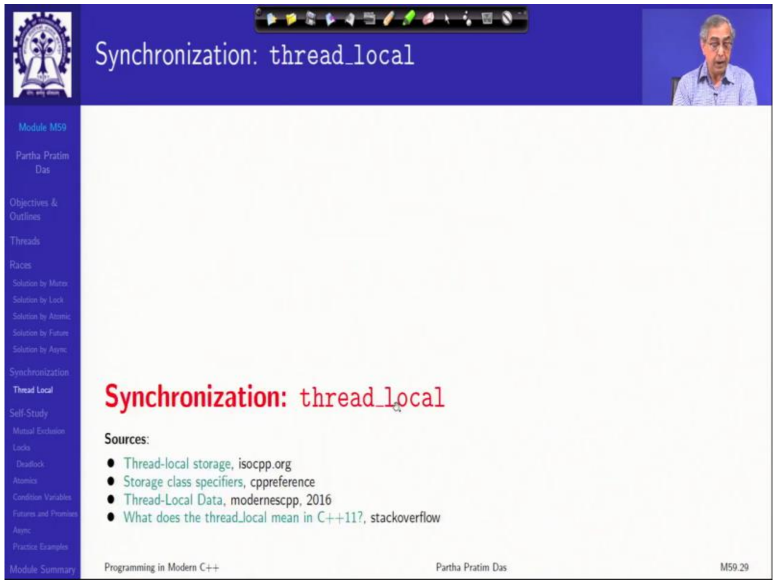
(Refer Slide Time: 24:31)



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.

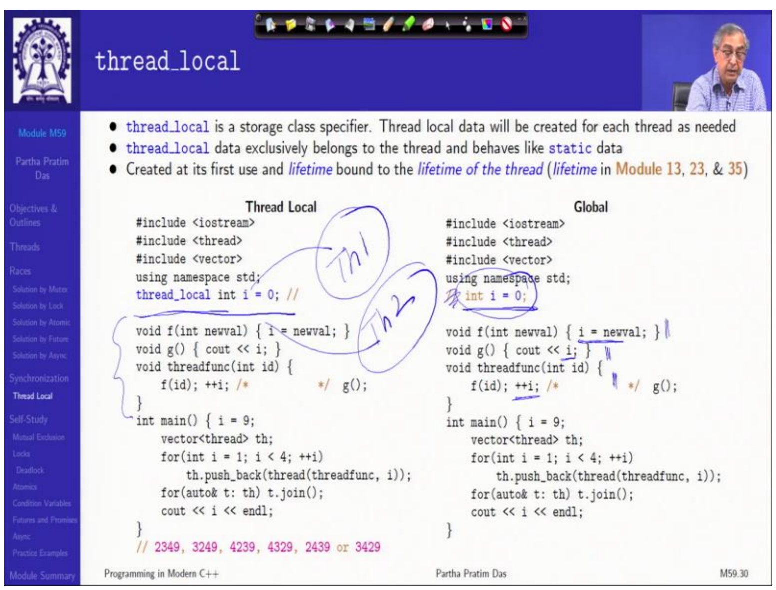
And we have except for condition variable which is particularly used for deadlock prevention, all other five we have seen examples of. And at the end I have given self-study module for you to go through each one of them and with sample programs attached with them.

(Refer Slide Time: 25:25)



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)



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 from before 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.

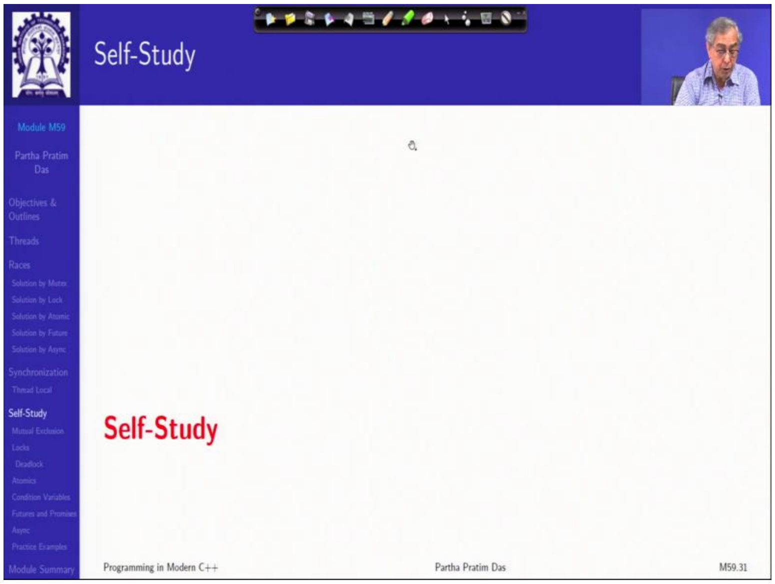
We have dynamic lifetime we choose our managers. Similarly, we add another lifetime here, which is called thread_local. The thread is created and an execution is happening. So, naturally, if a trade is created, an execution is happening, the thread has its own stack has to have otherwise functions could 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)

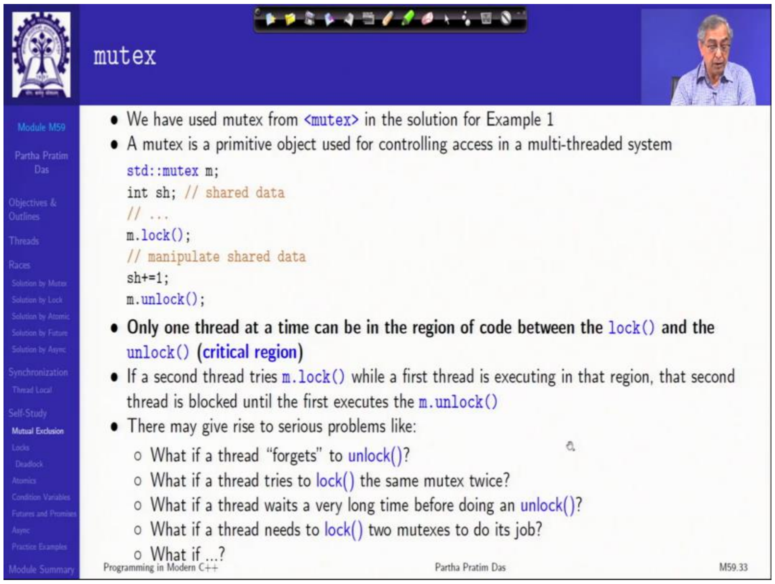


So, this is the short overview that I had for you.

(Refer Slide Time: 29:07)

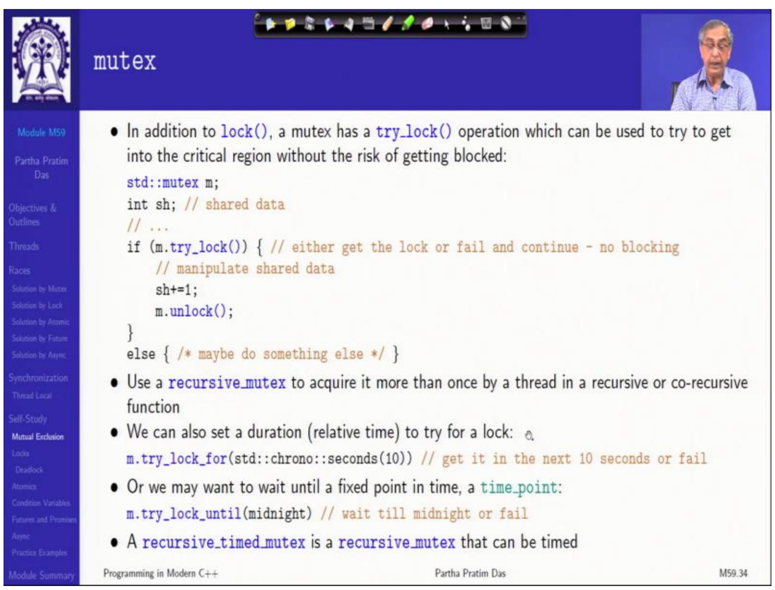
What I have now given is different slides I mean different sections for your self study.

(Refer Slide Time: 29:13)



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

(Refer Slide Time: 29:16)

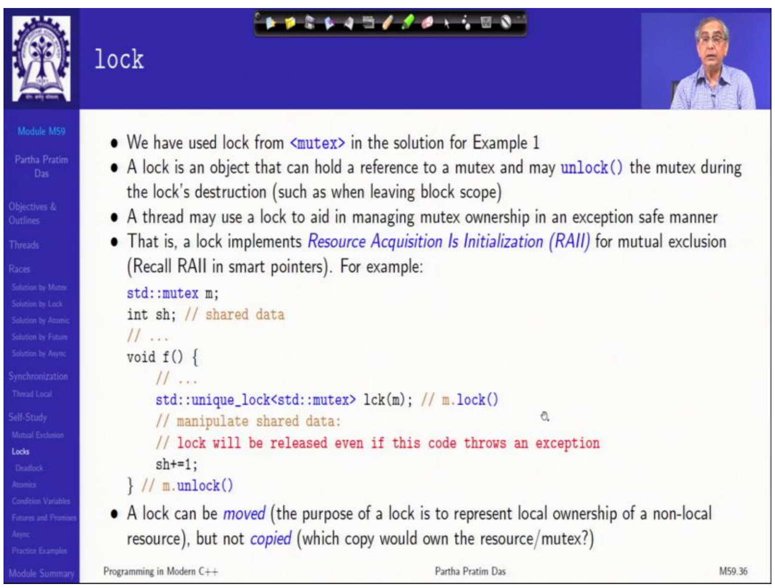


You have already seen the example use. And the specific features like besides lock and unlock it as a try_lock. So, learn what it is.

(Refer Slide Time: 29:24)

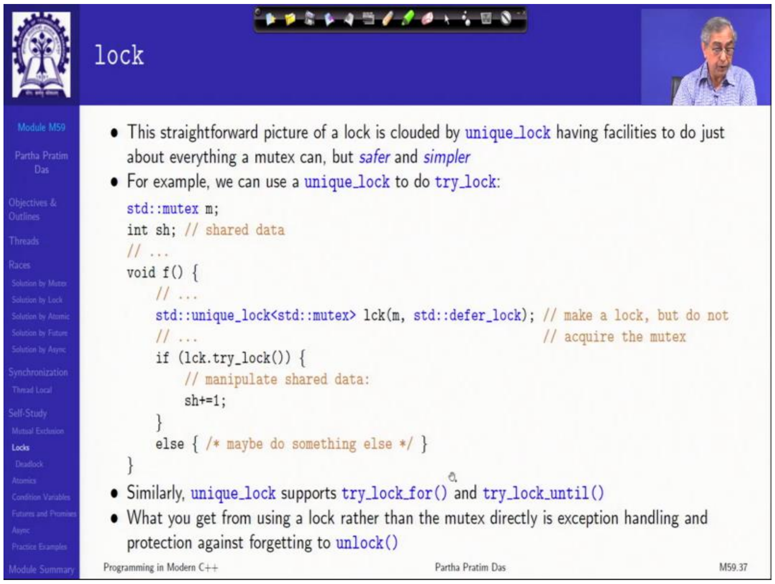
Then I have given more slides on the lock itself.

(Refer Slide Time: 29:27)



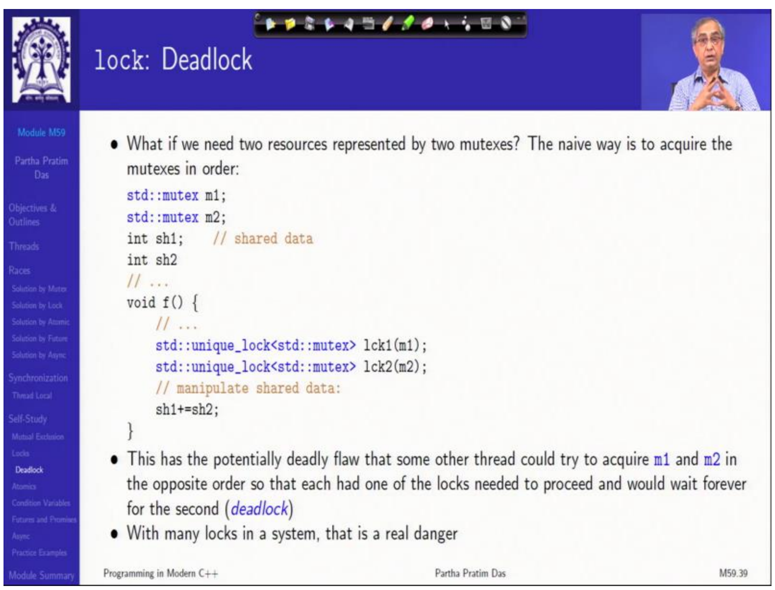
Which we have seen already how what that locks can be moved but they cannot be copied and so on so forth.

(Refer Slide Time: 29:38)



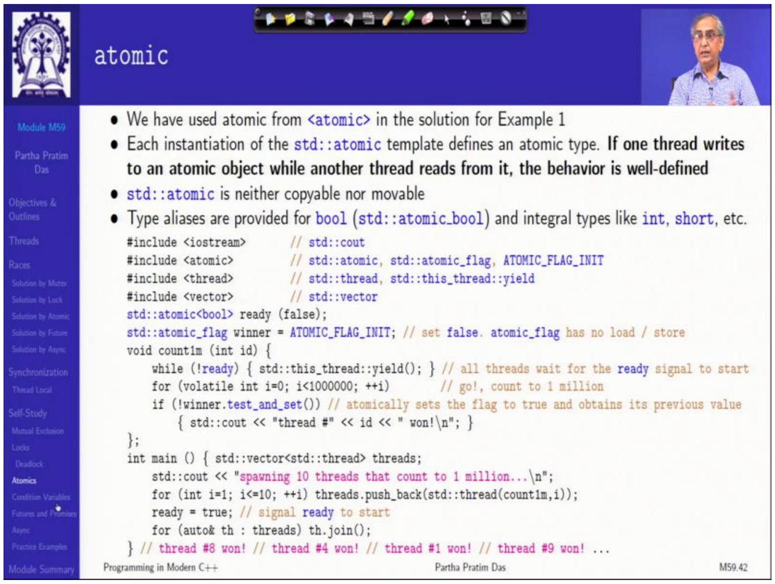
Very specifically, how locks can be used.

(Refer Slide Time: 29:43)



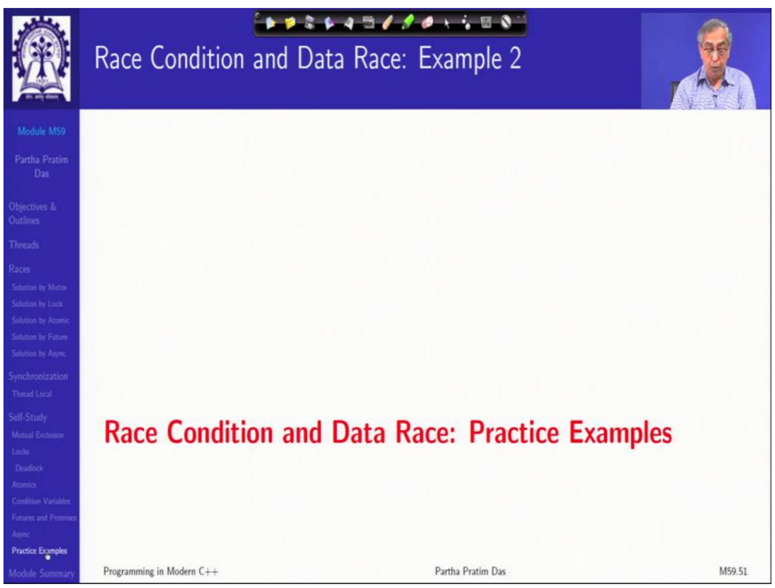
I mean what is the C++11 support mechanism to use locks in a way so that inherently deadlock can be prevented.

(Refer Slide Time: 29:53)



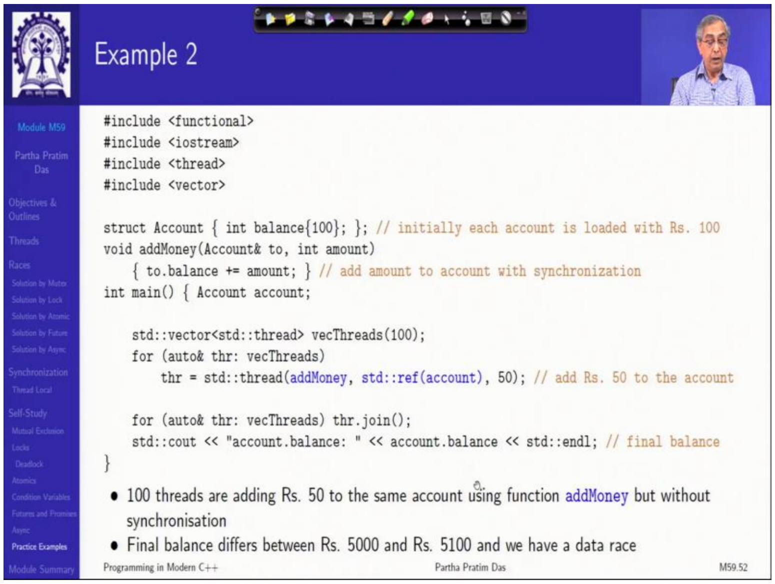
Then more on the atomic variables which I have given other examples condition variables, futures, async each one of them there are more.

(Refer Slide Time: 30:16)



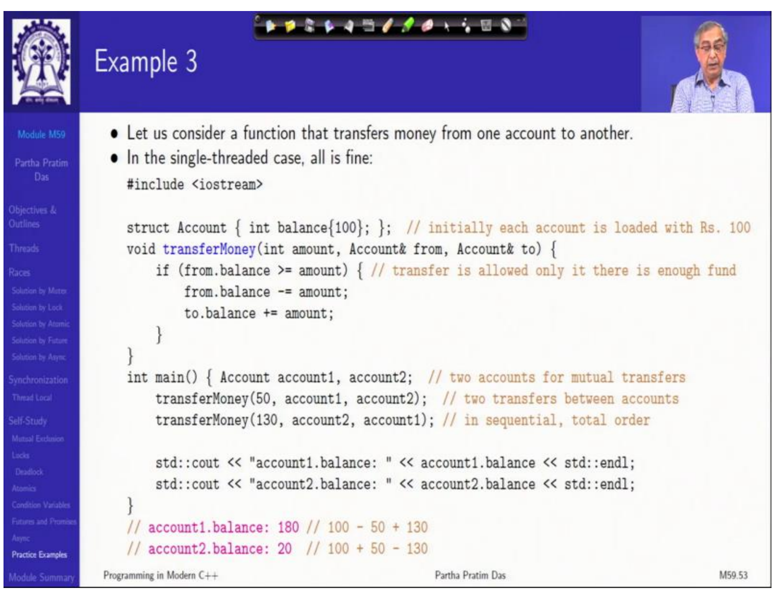
And then we have two more examples.

(Refer Slide Time: 30:09)



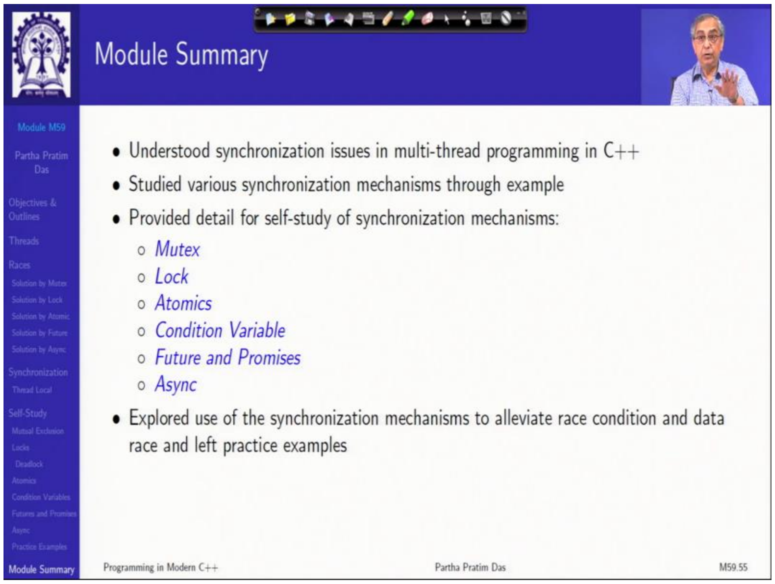
One in which, there is an account to which multiple threads are adding an amount. They are the same synchronization problem will arise. So, because multiple threads are, so multiple threads are actually updating their account. So, that is a situation and what you will have to do is to use those mechanisms 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)



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)



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.