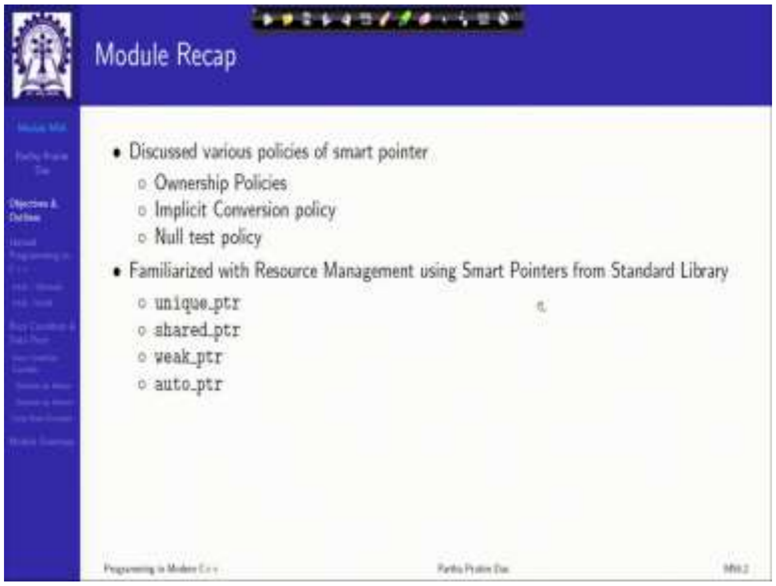
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture – 58
C++ and Beyond: Concurrency: Part 1

Welcome to programming in modern C++, we are going to discuss module 58.

(Refer Slide Time: 00:36)



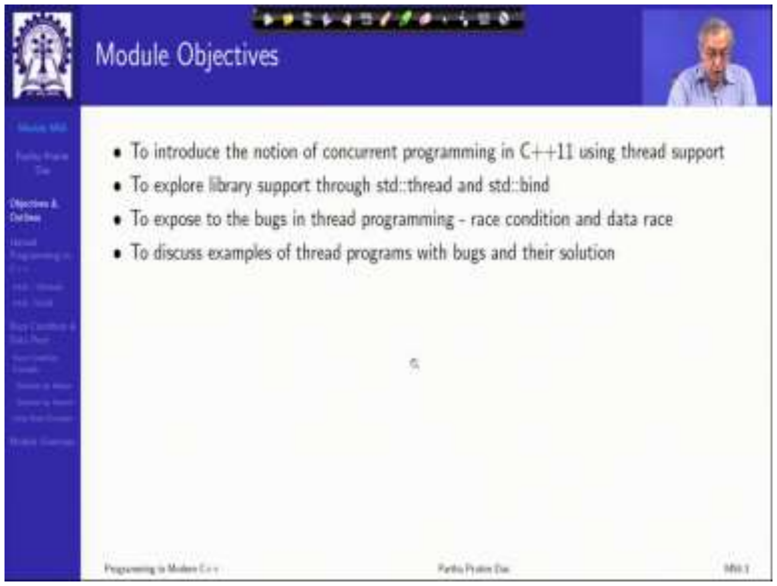
Module Recap

- Discussed various policies of smart pointer
  - Ownership Policies
  - Implicit Conversion policy
  - Null test policy
- Familiarized with Resource Management using Smart Pointers from Standard Library
  - `unique_ptr`
  - `shared_ptr`
  - `weak_ptr`
  - `auto_ptr`

Programming in Modern C++ Partha Pratim Das M58.2

In the last module, we have concluded discussions on smart pointers, their policies and we have familiarized with resource management using standard smart pointers from the library. Specifically, unique_ptr, shared_ptr and weak_ptr, auto_ptr is deprecated.

(Refer Slide Time: 00:59)



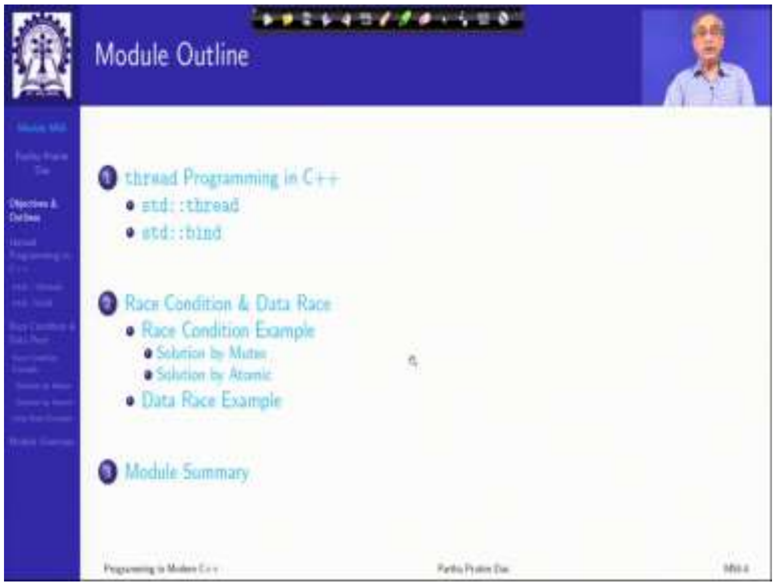
Module Objectives

- To introduce the notion of concurrent programming in C++11 using thread support
- To explore library support through std::thread and std::bind
- To expose to the bugs in thread programming - race condition and data race
- To discuss examples of thread programs with bugs and their solution

In this module, today, we are going to introduce the notion of concurrent programming in C++ using C++'s own threads support. So, for this you will need to have some understanding about concurrent programming, which I will assume that you have if you do not then you will have to read up other material to understand what is concurrent programming and what are its issues. Because that is an area which is not limited to C++ programming.

That is a general computer science knowledge about how do you program concurrently parallely and so, on. Now, we will specifically expose here to the library support through std::thread component and bind component. And discuss what are the, what are some of the common bugs in thread trade programming that can happen. Specifically, the race condition and data race. And we will discuss examples of thread programs which such bugs and look at some of their solutions.

(Refer Slide Time: 02:15)



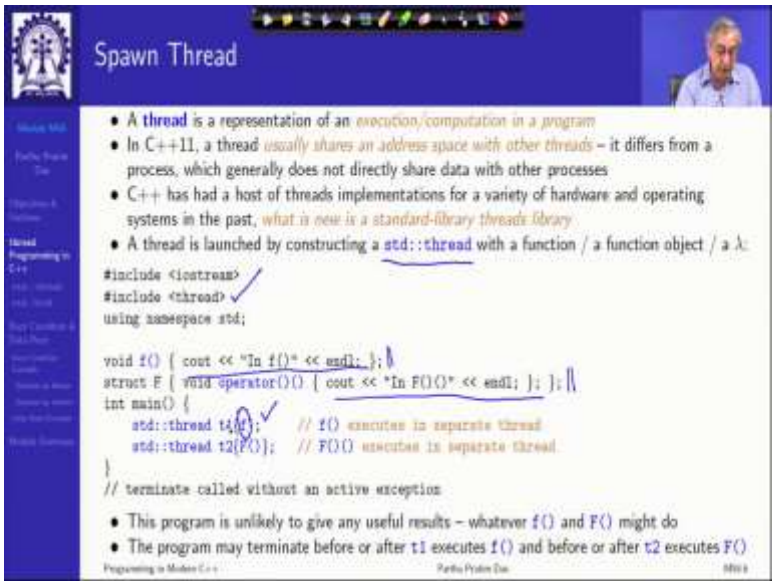
Module Outline

1. thread Programming in C++
   - std::thread
   - std::bind
2. Race Condition & Data Race
   - Race Condition Example
     - Solution by Mutex
     - Solution by Atomic
   - Data Race Example
3. Module Summary

Programming in Modern C++ Partha Pratim Das M46.4

So, this is the content for this module.

(Refer Slide Time: 02:25)



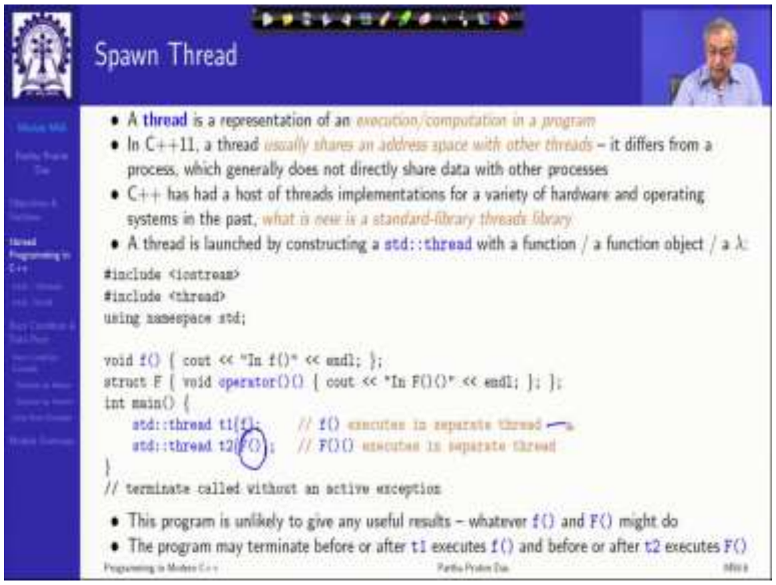
Spawn Thread

- A **thread** is a representation of an *execution/computation in a program*
- In C++11, a thread *usually shares an address space with other threads* – it differs from a process, which generally does not directly share data with other processes
- C++ has had a host of threads implementations for a variety of hardware and operating systems in the past, *what is new is a standard-library threads library*
- A thread is launched by constructing a `std::thread` with a function / a function object / a λ:

```
#include <iostream>
#include <thread>
using namespace std;

void f() { cout << "In f()" << endl; };
struct F { void operator()() { cout << "In F()()" << endl; }; };
int main() {
    std::thread t1(f);     // f() executes in separate thread
    std::thread t2(F());   // F()() executes in separate thread
}
// terminate called without an active exception
```

- This program is unlikely to give any useful results – whatever f() and F() might do
- The program may terminate before or after t1 executes f() and before or after t2 executes F()

Programming in Modern C++ Partha Pratim Das M46.6



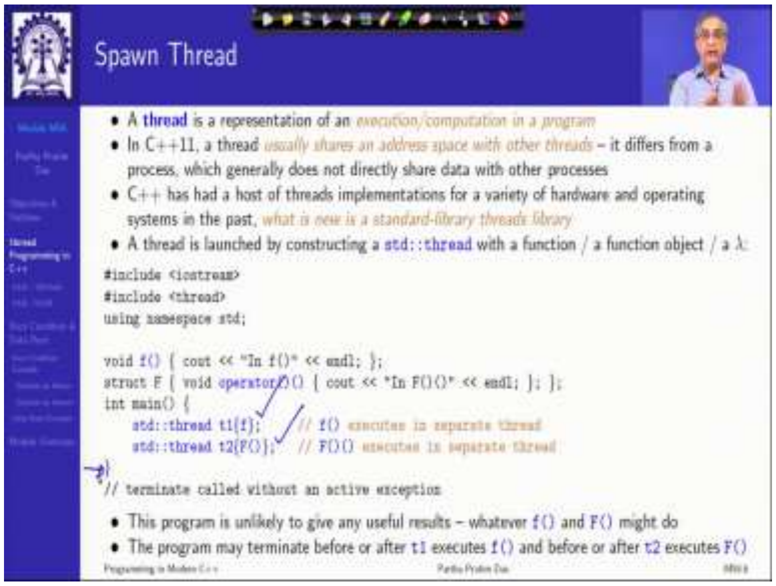
Spawn Thread

- A **thread** is a representation of an *execution/computation in a program*
- In C++11, a thread *usually shares an address space with other threads* – it differs from a process, which generally does not directly share data with other processes
- C++ has had a host of threads implementations for a variety of hardware and operating systems in the past, *what is new is a standard-library threads library*
- A thread is launched by constructing a `std::thread` with a function / a function object / a λ:

```
#include <iostream>
#include <thread>
using namespace std;

void f() { cout << "In f()" << endl; };
struct F { void operator()() { cout << "In F()()" << endl; }; };
int main() {
    std::thread t1{f};      // f() executes in separate thread
    std::thread t2{F()};    // F()() executes in separate thread
}
// terminate called without an active exception
```

- This program is unlikely to give any useful results – whatever `f()` and `F()` might do
- The program may terminate before or after `t1` executes `f()` and before or after `t2` executes `F()`

Programming in Modern C++ Partha Pratim Das M40.8



Spawn Thread

- A **thread** is a representation of an *execution/computation in a program*
- In C++11, a thread *usually shares an address space with other threads* – it differs from a process, which generally does not directly share data with other processes
- C++ has had a host of threads implementations for a variety of hardware and operating systems in the past, *what is new is a standard-library threads library*
- A thread is launched by constructing a `std::thread` with a function / a function object / a λ:

```
#include <iostream>
#include <thread>
using namespace std;

void f() { cout << "In f()" << endl; };
struct F { void operator()() { cout << "In F()()" << endl; }; };
int main() {
    std::thread t1{f};      // f() executes in separate thread
    std::thread t2{F()};    // F()() executes in separate thread
}
// terminate called without an active exception
```

- This program is unlikely to give any useful results – whatever `f()` and `F()` might do
- The program may terminate before or after `t1` executes `f()` and before or after `t2` executes `F()`

Programming in Modern C++ Partha Pratim Das M40.8

And let us start discussing with thread programming in C++. You know about processes in systems, like process is when we start a program it starts as a process and multiple processes run in the system at the same time. And the process, processes can share, exchange data by what is known as inter process communication. In so, processes are can be concurrent, they can be parallel and so, on.

Now, thread is a lightweight process, a multiple threads run within one process. So, the advantage that threads have that threads can communicate also through shared variables, shared data, because they are within one process space. Which processes cannot do because they have to protect the boundary of the processes data from corruption from other processes and so on. But the threads, all threads are within one process and they can communicate to share data.

They can also do inter process communication kind of mechanism, but they are necessarily lightweight processes. And therefore, in the recent past, recent in the sense of say last two decades, concurrent programming using threads have become a dominant feature. So, they the naturally being in the address space of a single process, they shared address space with other threads and can communicate freely and for decades the C programmer, C++ programmers have been doing concurrent programming, multi threaded programming as it is said using several different libraries.

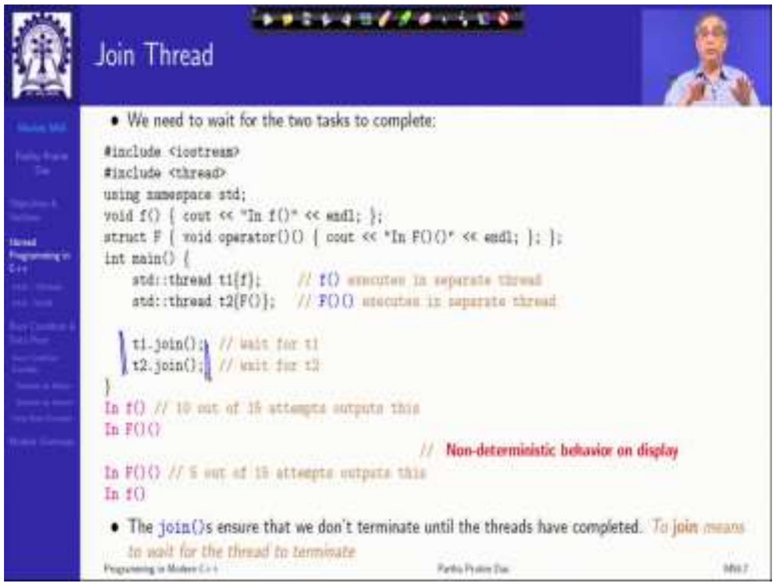
Predominantly the POSIX library which is common on the Linux platform or say the windows thread library then there are several others like boost and so on so forth. So, in that context, what is special about C++11 in the context of thread programming? The special thing is now, C++11 onwards, the library, standard library itself provides complete support for multithreaded programming. You do not have to use a third party library like the POSIX library or any other.

So, using the primary component which is the thread component in the library, you can launch a thread with a function, a function object, or even with a lambda. So you include the, that component. And suppose I have a function, small f here, and you have a function object, capital F here, and they do not do anything, they just kind of print a message, the overloaded operator just prints a message. And this is how you invoke the thread.

So, you construct a thread t1 with the parameter f, the function parameter. I am using the C++ style of initialization because that is what you should be comfortable with. Similarly, it could thread t2 could construct a functor object to initialize with. And as soon as that thread is constructed, as in here, it starts executing. Now, if you just run this program, copy, paste and run this program, you will see that you are not probably not seeing the outputs is not unlikely that you will see any result.

The reason is, this thread gets started, as soon as constructed, this thread get started as soon as constructed. And then main comes to an end. So main, now, I have created a thread, so it is done. So, the main will complete, but main function is the main thread in the process that you are running. So, once main completes, these threads are automatically terminated. So, they will not show you any effect. So, what we will have to do for an effective thread programming, we will have to tell main the main thread that wait these threads are still working do not terminate right now wait for them to finish.

(Refer Slide Time: 06:57)



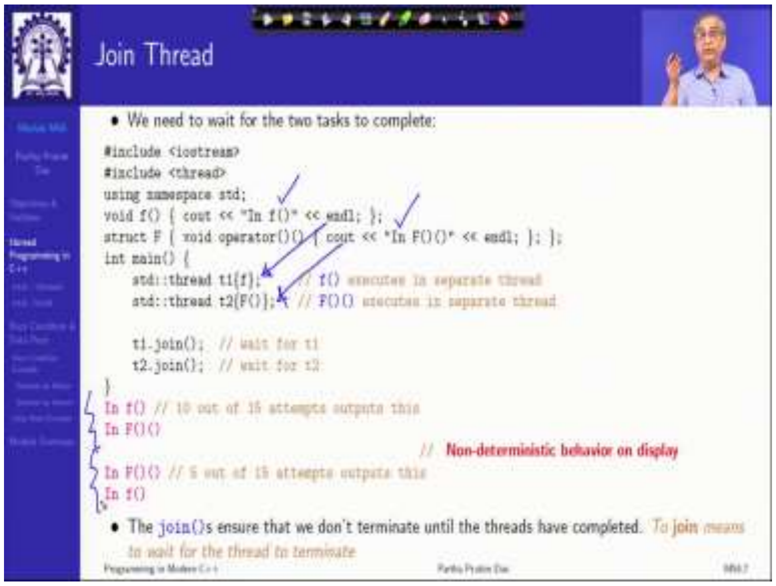
Join Thread

- We need to wait for the two tasks to complete:

```
#include <iostream>
#include <thread>
using namespace std;
void f() { cout << "In f()" << endl; };
struct F { void operator()() { cout << "In F()()" << endl; }; };
int main() {
    std::thread t1{f};      // f() executes in separate thread
    std::thread t2{F()};    // F()() executes in separate thread

    t1.join();  // wait for t1
    t2.join();  // wait for t2
}
In f() // 10 out of 15 attempts outputs this
In F()()
                                        // Non-deterministic behavior on display
In F()() // 5 out of 15 attempts outputs this
In f()
```

- The join()s ensure that we don't terminate until the threads have completed. *To join means to wait for the thread to terminate*

Programming in Modern C++ Partha Pratim Das



Join Thread

- We need to wait for the two tasks to complete:

```
#include <iostream>
#include <thread>
using namespace std;
void f() { cout << "In f()" << endl; };
struct F { void operator()() { cout << "In F()()" << endl; }; };
int main() {
    std::thread t1{f};      // f() executes in separate thread
    std::thread t2{F()};    // F()() executes in separate thread

    t1.join();  // wait for t1
    t2.join();  // wait for t2
}
In f() // 10 out of 15 attempts outputs this
In F()()
                                        // Non-deterministic behavior on display
In F()() // 5 out of 15 attempts outputs this
In f()
```

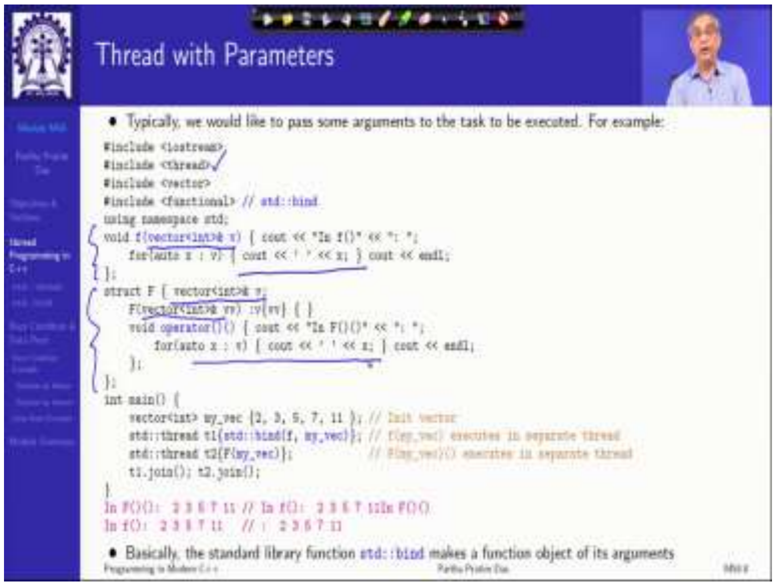
- The join()s ensure that we don't terminate until the threads have completed. *To join means to wait for the thread to terminate*

Programming in Modern C++ Partha Pratim Das

So, you say that by saying the thread variable dot join t1.join. So, t1 dot join tells me that well you wait till I finish. Similarly, t2.join says that. So, with this, now, the both threads will have to complete before main can actually terminate and therefore, you will see the result. This is, now the interesting thing is these are parallel these are concurrent executions are happening at the same time.

Even though t1 is constructed first and t2 is constructed next, and they start working as soon as they are constructed, it is not known in which order they will execute the corresponding cout. So, if you run it multiple times, in some cases, you will find that the function f1 is executing first. And in some cases, you will find the functor F is executing first. I tried fifteen consecutive times. And out of that ten times I got this and five times I got this. So, that is the indeterminism in terms of thread execution, that will always be there.

(Refer Slide Time: 08:14)



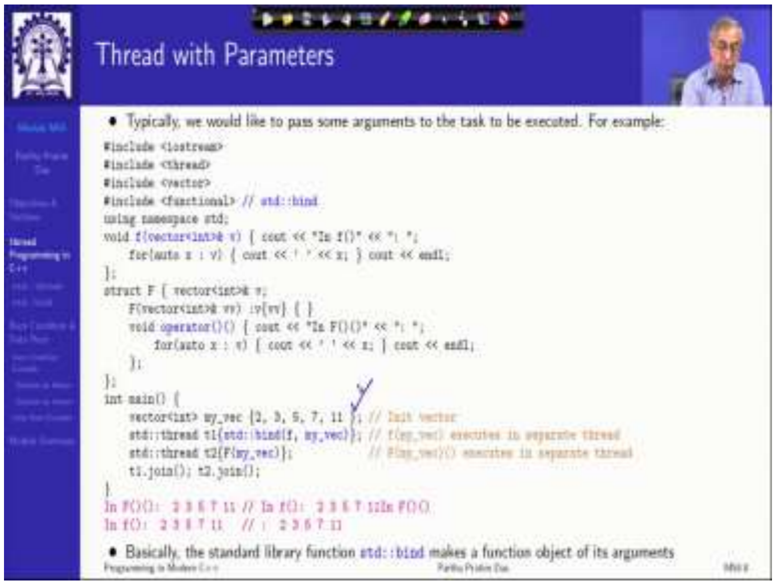
Thread with Parameters

- Typically, we would like to pass some arguments to the task to be executed. For example:

```
#include <iostream>
#include <thread>
#include <vector>
#include <functional> // std::bind
using namespace std;
void f(vector<int>& v) { cout << "In f()" << ": ";
    for(auto x : v) { cout << ' ' << x; } cout << endl;
};
struct F { vector<int>& v;
    F(vector<int>& vv) :v{vv} { }
    void operator()() { cout << "In F()()" << ": ";
        for(auto x : v) { cout << ' ' << x; } cout << endl;
    };
};
int main() {
    vector<int> my_vec {2, 3, 5, 7, 11 }; // Init vector
    std::thread t1{std::bind(f, my_vec)}; // f(my_vec) executes in separate thread
    std::thread t2{F(my_vec)};            // F(my_vec)() executes in separate thread
    t1.join(); t2.join();
}
In F()(): 2 3 5 7 11 // In f(): 2 3 5 7 11In F()()
In f(): 2 3 5 7 11   // : 2 3 5 7 11
```

- Basically, the standard library function std::bind makes a function object of its arguments

Programming in Modern C++ Partha Pratim Das M44.8



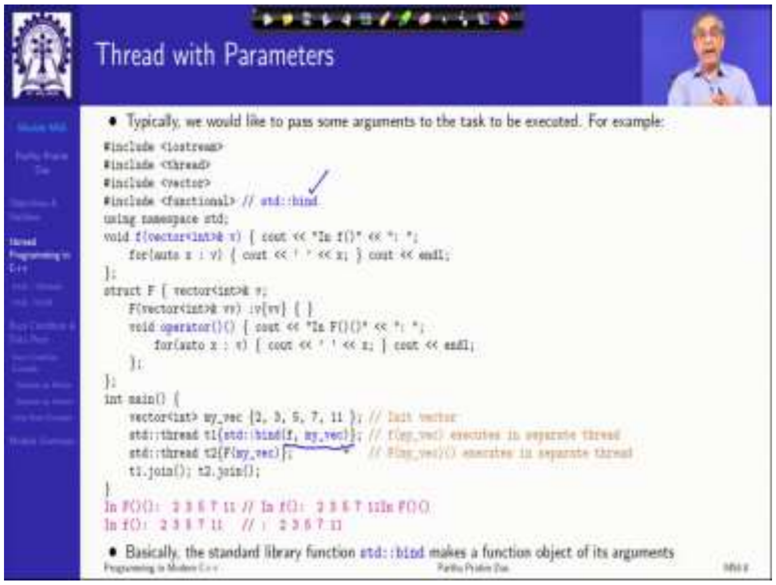
Thread with Parameters

- Typically, we would like to pass some arguments to the task to be executed. For example:

```
#include <iostream>
#include <thread>
#include <vector>
#include <functional> // std::bind
using namespace std;
void f(vector<int>& v) { cout << "In f()" << ": ";
    for(auto x : v) { cout << ' ' << x; } cout << endl;
};
struct F { vector<int>& v;
    F(vector<int>& vv) :v{vv} { }
    void operator()() { cout << "In F()()" << ": ";
        for(auto x : v) { cout << ' ' << x; } cout << endl;
    };
};
int main() {
    vector<int> my_vec {2, 3, 5, 7, 11 }; // Init vector
    std::thread t1{std::bind(f, my_vec)}; // f(my_vec) executes in separate thread
    std::thread t2{F(my_vec)};            // F(my_vec)() executes in separate thread
    t1.join(); t2.join();
}
In F()(): 2 3 5 7 11 // In f(): 2 3 5 7 11In F()()
In f(): 2 3 5 7 11   // : 2 3 5 7 11
```

- Basically, the standard library function std::bind makes a function object of its arguments

Programming in Modern C++ Partha Pratim Das M44.8



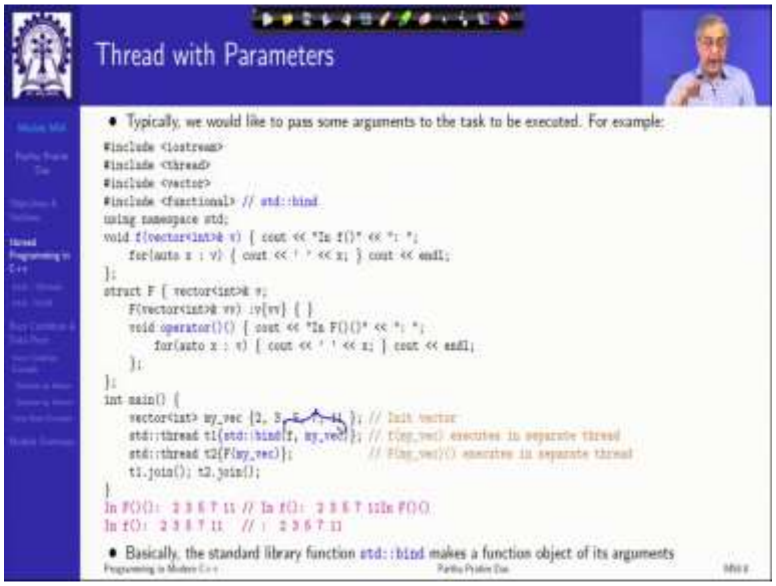
Thread with Parameters

- Typically, we would like to pass some arguments to the task to be executed. For example:

```
#include <iostream>
#include <thread>
#include <vector>
#include <functional> // std::bind
using namespace std;
void f(vector<int>& v) { cout << "In f()" << ": ";
    for(auto x : v) { cout << ' ' << x; } cout << endl;
};
struct F { vector<int>& v;
    F(vector<int>& vv) :v{vv} { }
    void operator()() { cout << "In F()()" << ": ";
        for(auto x : v) { cout << ' ' << x; } cout << endl;
    };
};
int main() {
    vector<int> my_vec {2, 3, 5, 7, 11 }; // Init vector
    std::thread t1{std::bind(f, my_vec)}; // f(my_vec) executes in separate thread
    std::thread t2{F(my_vec)};             // F(my_vec)() executes in separate thread
    t1.join(); t2.join();
}
In F()():  2 3 5 7 11 // In f():  2 3 5 7 11In F()()
In f():  2 3 5 7 11   // :  2 3 5 7 11
```

- Basically, the standard library function `std::bind` makes a function object of its arguments

Programming in Modern C++ Partha Pratim Das M44.8



Thread with Parameters

- Typically, we would like to pass some arguments to the task to be executed. For example:

```
#include <iostream>
#include <thread>
#include <vector>
#include <functional> // std::bind
using namespace std;
void f(vector<int>& v) { cout << "In f()" << ": ";
    for(auto x : v) { cout << ' ' << x; } cout << endl;
};
struct F { vector<int>& v;
    F(vector<int>& vv) :v{vv} { }
    void operator()() { cout << "In F()()" << ": ";
        for(auto x : v) { cout << ' ' << x; } cout << endl;
    };
};
int main() {
    vector<int> my_vec {2, 3, 5, 7, 11 }; // Init vector
    std::thread t1{std::bind(f, my_vec)}; // f(my_vec) executes in separate thread
    std::thread t2{F(my_vec)};             // F(my_vec)() executes in separate thread
    t1.join(); t2.join();
}
In F()():  2 3 5 7 11 // In f():  2 3 5 7 11In F()()
In f():  2 3 5 7 11   // :  2 3 5 7 11
```

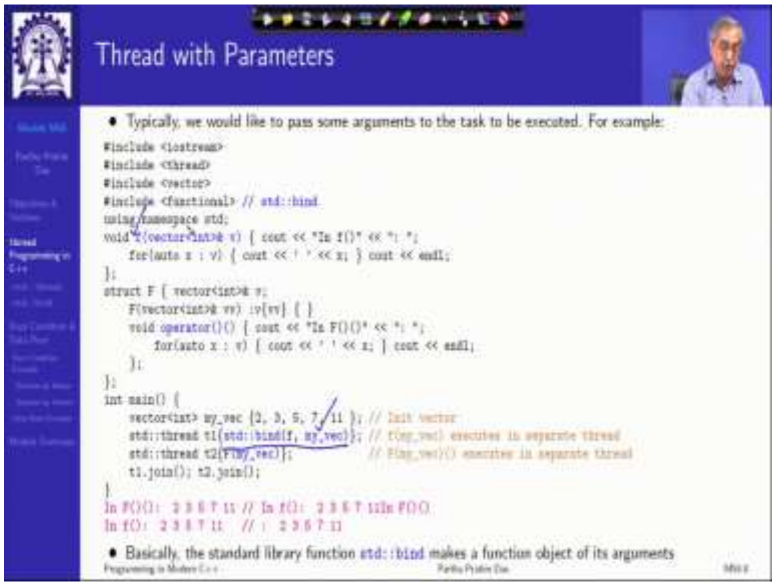
- Basically, the standard library function `std::bind` makes a function object of its arguments

Programming in Modern C++ Partha Pratim Das M44.8

Now this is this is fine. As long as you are just printing miscellaneous, but obviously the function will need to have parameters. So, let us assume that our thread that the function of the functor that it tries to it is trying to execute there is a parameter. So, the function has a parameter, let us say a vector. And what it does, it kind of prints the vector with a range for kind of construct. Similarly, the functor has a vector as a local variable as a data member.

And it is constructed with that, and the functor will also do that, it will print that the vector. I have constructed a vector. So, now the question is how do I tell the thread that not only this function, but I also need to give this parameter. So for this, you have a special template called bind. So, bind basically takes a function and one or more parameters of that function, puts them together into a function object and then passes that to wherever it is required. So, it creates a functor instance from the function or the functor or the lambda and the parameters.

(Refer Slide Time: 09:46)



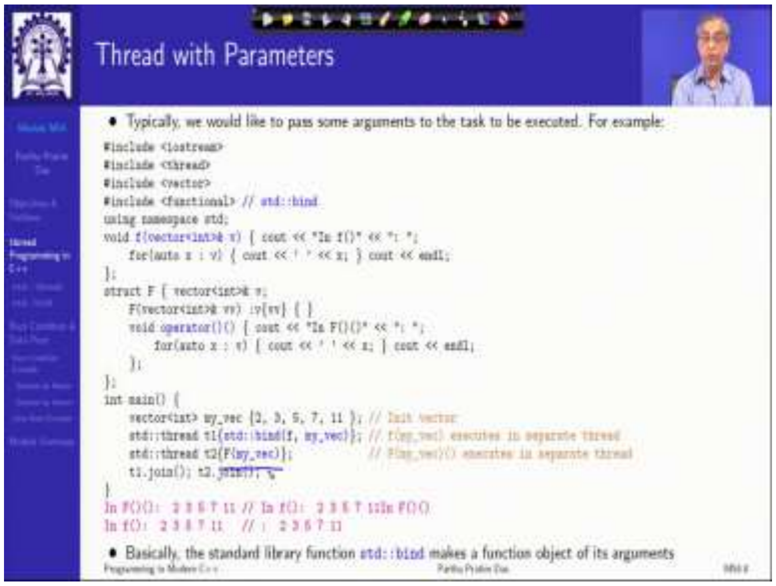
Thread with Parameters

- Typically, we would like to pass some arguments to the task to be executed. For example:

```
#include <iostream>
#include <thread>
#include <vector>
#include <functional> // std::bind
using namespace std;
void f(vector<int>& v) { cout << "In f()" << ": ";
    for(auto x : v) { cout << ' ' << x; } cout << endl;
};
struct F { vector<int>& v;
    F(vector<int>& vv) :v{vv} { }
    void operator()() { cout << "In F()()" << ": ";
        for(auto x : v) { cout << ' ' << x; } cout << endl;
    };
};
int main() {
    vector<int> my_vec {2, 3, 5, 7, 11 }; // Init vector
    std::thread t1{std::bind(f, my_vec)}; // f(my_vec) executes in separate thread
    std::thread t2{F(my_vec)};            // F(my_vec)() executes in separate thread
    t1.join(); t2.join();
}
In F()(): 2 3 5 7 11 // In f(): 2 3 5 7 11In F()()
In f(): 2 3 5 7 11   // : 2 3 5 7 11
```

- Basically, the standard library function std::bind makes a function object of its arguments

Programming in Modern C++ Partha Pratim Das M44.8



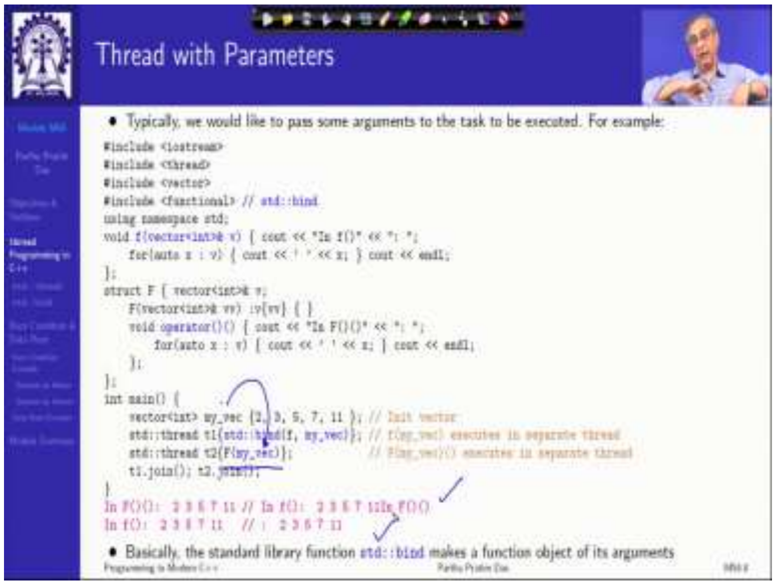
Thread with Parameters

- Typically, we would like to pass some arguments to the task to be executed. For example:

```
#include <iostream>
#include <thread>
#include <vector>
#include <functional> // std::bind
using namespace std;
void f(vector<int>& v) { cout << "In f()" << ": ";
    for(auto x : v) { cout << ' ' << x; } cout << endl;
};
struct F { vector<int>& v;
    F(vector<int>& vv) :v{vv} { }
    void operator()() { cout << "In F()()" << ": ";
        for(auto x : v) { cout << ' ' << x; } cout << endl;
    };
};
int main() {
    vector<int> my_vec {2, 3, 5, 7, 11 }; // Init vector
    std::thread t1{std::bind(f, my_vec)}; // f(my_vec) executes in separate thread
    std::thread t2{F(my_vec)};            // F(my_vec)() executes in separate thread
    t1.join(); t2.join();
}
In F()(): 2 3 5 7 11 // In f(): 2 3 5 7 11In F()()
In f(): 2 3 5 7 11   // : 2 3 5 7 11
```

- Basically, the standard library function std::bind makes a function object of its arguments

Programming in Modern C++ Partha Pratim Das M44.8



Thread with Parameters

- Typically, we would like to pass some arguments to the task to be executed. For example:

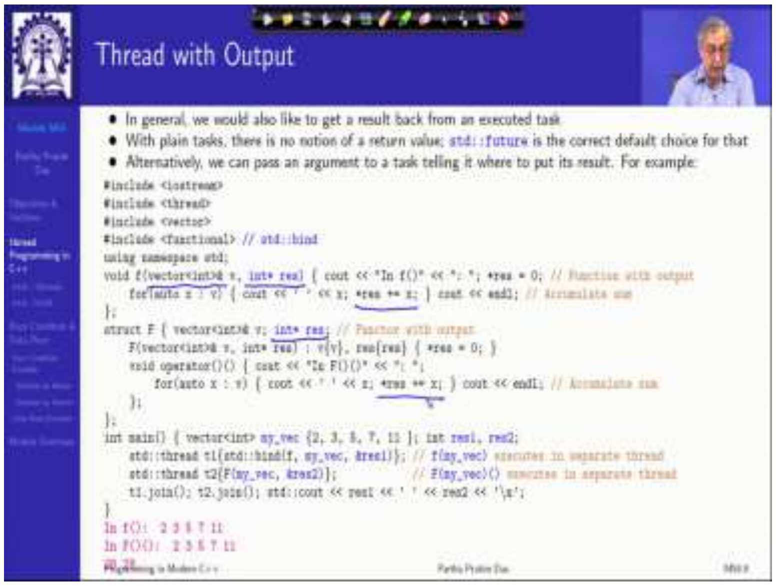
```
#include <iostream>
#include <thread>
#include <vector>
#include <functional> // std::bind
using namespace std;
void f(vector<int>& v) { cout << "In f()" << ": ";
    for(auto x : v) { cout << ' ' << x; } cout << endl;
};
struct F { vector<int>& v;
    F(vector<int>& vv) :v{vv} { }
    void operator()() { cout << "In F()()" << ": ";
        for(auto x : v) { cout << ' ' << x; } cout << endl;
    };
};
int main() {
    vector<int> my_vec {2, 3, 5, 7, 11 }; // Init vector
    std::thread t1{std::bind(f, my_vec)}; // f(my_vec) executes in separate thread
    std::thread t2{F(my_vec)};            // F(my_vec)() executes in separate thread
    t1.join(); t2.join();
}
In F()(): 2 3 5 7 11 // In f(): 2 3 5 7 11In F()()
In f(): 2 3 5 7 11   // : 2 3 5 7 11
```

- Basically, the standard library function `std::bind` makes a function object of its arguments

Programming in Modern C++ Partha Pratim Das M44.8

So, the bind is successfully used in this case, in the first case, to get the function and the parameter to pass to the thread. So, now when the thread runs, it is invoking f with the parameter that my_vec that we have passed. In the second case, since F is already a functor, we do not need a bind, all that I need to do is to construct an instance of F with the my_vec, which I do here. And you can see the result that that happens. Now, you will also see that if you run it multiple times, you will also see, in some cases, the functor runs first. On some cases, the the function runs first, it is indeterminable.

(Refer Slide Time: 10:40)

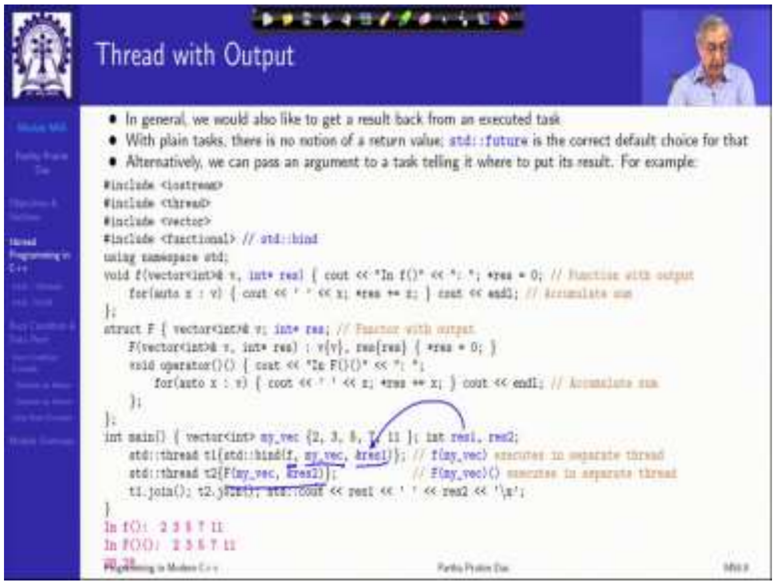


Thread with Output

- In general, we would also like to get a result back from an executed task
- With plain tasks, there is no notion of a return value; `std::future` is the correct default choice for that
- Alternatively, we can pass an argument to a task telling it where to put its result. For example:

```
#include <iostream>
#include <thread>
#include <vector>
#include <functional> // std::bind
using namespace std;
void f(vector<int>& v, int* res) { cout << "In f()" << ": "; *res = 0; // Function with output
    for(auto x : v) { cout << ' ' << x; *res += x; } cout << endl; // Accumulate sum
};
struct F { vector<int>& v; int* res; // Functor with output
    F(vector<int>& v, int* res) : v{v}, res{res} { *res = 0; }
    void operator()() { cout << "In F()()" << ": ";
        for(auto x : v) { cout << ' ' << x; *res += x; } cout << endl; // Accumulate sum
    };
};
int main() { vector<int> my_vec {2, 3, 5, 7, 11 }; int res1, res2;
    std::thread t1{std::bind(f, my_vec, &res1)}; // f(my_vec) executes in separate thread
    std::thread t2{F(my_vec, &res2)};            // F(my_vec)() executes in separate thread
    t1.join(); t2.join(); std::cout << res1 << ' ' << res2 << '\n';
}
In f(): 2 3 5 7 11
In F()(): 2 3 5 7 11
28 28
```

Programming in Modern C++ Partha Pratim Das M44.9



Thread with Output

- In general, we would also like to get a result back from an executed task
- With plain tasks, there is no notion of a return value; std::future is the correct default choice for that
- Alternatively, we can pass an argument to a task telling it where to put its result. For example:

```
#include <iostream>
#include <thread>
#include <vector>
#include <functional> // std::bind
using namespace std;
void f(vector<int>& v, int* res) { cout << "In f()" << ": "; *res = 0; // Function with output
    for(auto x : v) { cout << ' ' << x; *res += x; } cout << endl; // Accumulate sum
};
struct F { vector<int>& v; int* res; // Functor with output
    F(vector<int>& v, int* res) : v{v}, res{res} { *res = 0; }
    void operator()() { cout << "In F()()" << ": ";
        for(auto x : v) { cout << ' ' << x; *res += x; } cout << endl; // Accumulate sum
    };
};
int main() { vector<int> my_vec {2, 3, 5, 7, 11 }; int res1, res2;
    std::thread t1{std::bind(f, my_vec, &res1)}; // f(my_vec) executes in separate thread
    std::thread t2{F(my_vec, &res2)};           // F(my_vec)() executes in separate thread
    t1.join(); t2.join(); std::cout << res1 << ' ' << res2 << '\n';
}
```

```
In f(): 2 3 5 7 11
In F()(): 2 3 5 7 11
28 28
```

Programming in Modern C++ Partha Pratim Das M46.8

Now, so we have the trade execution, we have the input. So, now the question is how do you get output. So, there are specific mechanisms to get output, though if a as a plain task, there is no notion of return value, it is just an execution in the thread. So, it is not like a function which you wherein you normally expect there will be a return value. This is a, this is a task, so, you just go and do that task, execute that function, execute that lambda and so on.

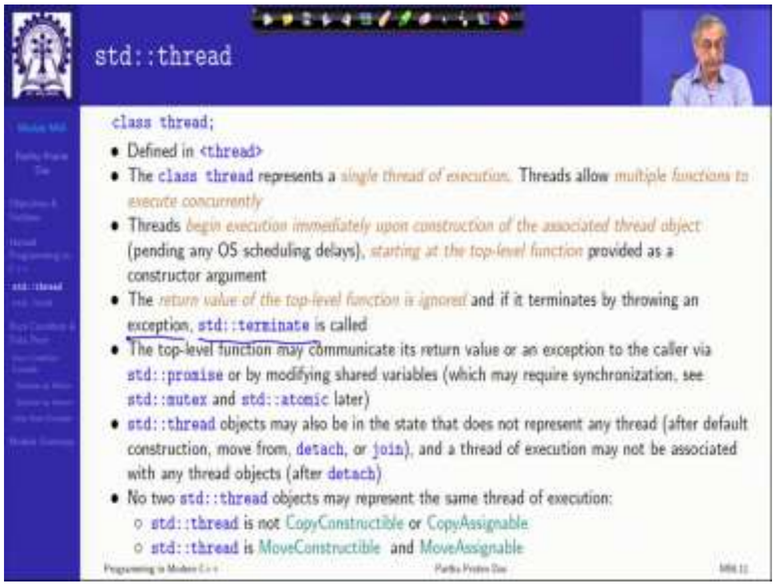
But there are special mechanisms of that, which we will discuss in the next module. But what you can always do is to get the output you can pass some parameters, as either a pointer parameter or you can pass a reference parameter and get the output through that. So, let us say we pass the vector as an input, and we pass a pointer to res. Where in res, we actually compute the sum of the vector elements.

Similarly, here we introduce another data member in the functor and compute the add summation. So, what now happens is, if I in, I have to invoke the function, I have to bind the function f, the first parameter my_vec and a second parameter, which is the result. The result is expected as a pointer to integer so, I have to pass the address of that. So, this is very typical of how you call a function.

You bind that function is created and you invoke t1 with it. Similarly, a functor is directly created in capital F. And if you now run it, you will not only be able to see what the, the list of vector elements are, but after both of them have joined, that is finished, you can print the results and you can see that the results are appearing here.

So, we have more or less the basic tasks that a function needs to do or function needs to do, we can do that using the thread, using concurrent executions.

(Refer Slide Time: 12:53)



std::thread

`class thread;`

- Defined in `<thread>`
- The `class thread` represents a *single thread of execution*. Threads allow *multiple functions to execute concurrently*
- Threads *begin execution immediately upon construction of the associated thread object* (pending any OS scheduling delays), *starting at the top-level function* provided as a constructor argument
- The *return value of the top-level function is ignored* and if it terminates by throwing an exception, `std::terminate` is called
- The top-level function may communicate its return value or an exception to the caller via `std::promise` or by modifying shared variables (which may require synchronization, see `std::mutex` and `std::atomic` later)
- `std::thread` objects may also be in the state that does not represent any thread (after default construction, move from, `detach`, or `join`), and a thread of execution may not be associated with any thread objects (after `detach`)
- No two `std::thread` objects may represent the same thread of execution:
  - `std::thread` is not CopyConstructible or CopyAssignable
  - `std::thread` is MoveConstructible and MoveAssignable

Programming in Modern C++ Partha Pratim Das

std::thread

`class thread;`

- Defined in `<thread>`
- The `class thread` represents a *single thread of execution*. Threads allow *multiple functions to execute concurrently*
- Threads *begin execution immediately upon construction of the associated thread object* (pending any OS scheduling delays), *starting at the top-level function* provided as a constructor argument
- The *return value of the top-level function is ignored* and if it terminates by throwing an exception, `std::terminate` is called
- The top-level function may communicate its return value or an exception to the caller via `std::promise` or by modifying shared variables (which may require synchronization, see `std::mutex` and `std::atomic` later)
- `std::thread` objects may also be in the state that does not represent any thread (after default construction, move from, `detach`, or `join`), and a thread of execution may not be associated with any thread objects (after `detach`)
- No two `std::thread` objects may represent the same thread of execution:
  - `std::thread` is not CopyConstructible or CopyAssignable
  - `std::thread` is MoveConstructible and MoveAssignable

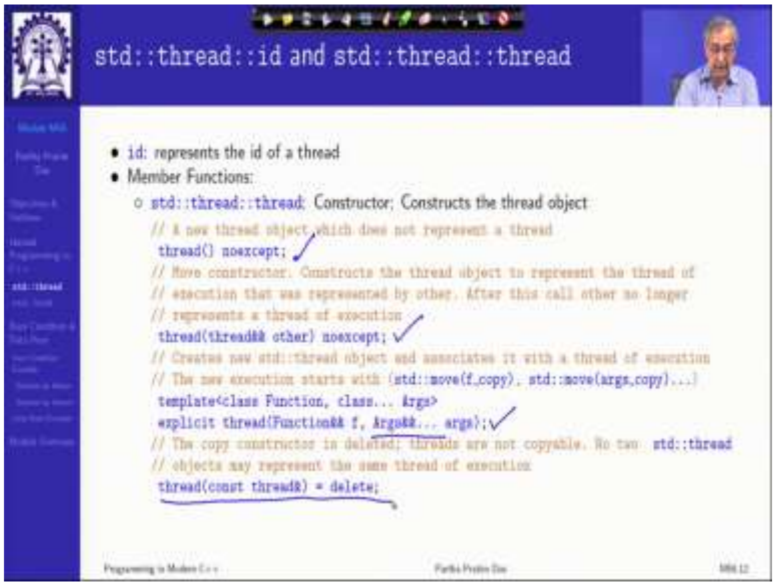
Programming in Modern C++ Partha Pratim Das

Let us take a little, more elaborate look into what does std::thread has. It is defined in the thread component as a threat, it represents a single thread of execution and allows multiple functions to execute concurrently in multiple different threads. So, return value is typically ignored, as I have said, but if it terminates by throwing an exception, if it terminates by throwing an exception, then std::terminate is called, which means that the entire program is terminated.

So, if any of the threads throws an exception, then the entire program will be terminated. So, you will have to, control that within that. So, there are mechanisms for communication of the return value, which we will we will see subsequently. But the most important thing is that a thread may not always have a function associated with it to, to basically execute. So, the thread may be in a state or a thread object rather, I mean not thread.

Thread object may be in a state that it actually does not represent a thread of execution. So, this happens after a default construction, or you move the thread from the thread object to another thread object you detach, or you join when the task is completed. So, in this, this thread object is unique in terms of execution. That is no thread object can represent the same thread of execution. And interestingly, the thread objects are not copyable, they cannot be copy constructed or assigned, but they are movable.

(Refer Slide Time: 14:54)



std::thread::id and std::thread::thread

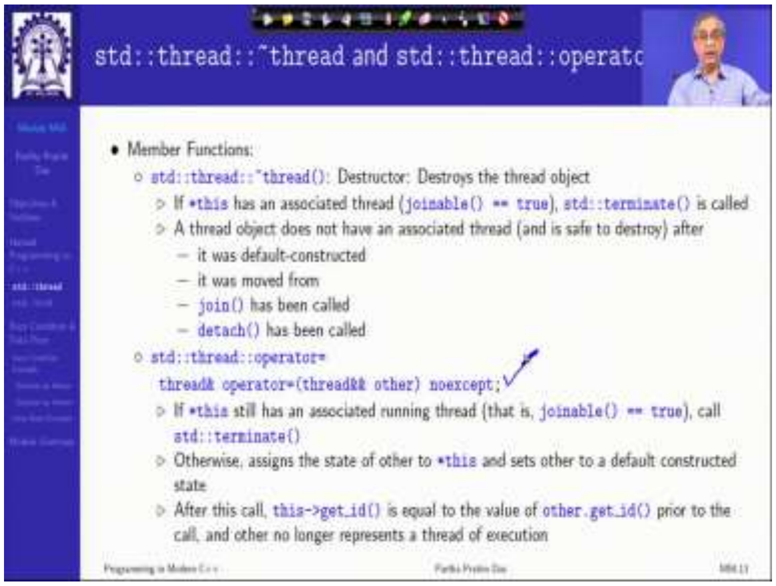
- id: represents the id of a thread
- Member Functions:
  - std::thread::thread: Constructor: Constructs the thread object

```
// A new thread object which does not represent a thread
thread() noexcept;
// Move constructor. Constructs the thread object to represent the thread of
// execution that was represented by other. After this call other no longer
// represents a thread of execution
thread(thread&& other) noexcept;
// Creates new std::thread object and associates it with a thread of execution
// The new execution starts with (std::move(f_copy), std::move(args_copy)...)
template<class Function, class... Args>
explicit thread(Function&& f, Args&&... args);
// The copy constructor is deleted; threads are not copyable. No two std::thread
// objects may represent the same thread of execution
thread(const thread&) = delete;
```

Programming in Modern C++ Partha Pratim Das M46.12

So if you look at the basic, primary structure of the thread class, you will see that there is a default constructor. There is a default constructor, there is a move constructor because it is moved constructible, there is a parameterised constructor, which is explicit. This is templatized, so that you can send, you can see that there is a variadic template being used here. And it has a, you can see that this is deleted, that is the copy construction is deleted. So, copy construction is not allowed. So, these are the basic properties.

(Refer Slide Time: 15:38)



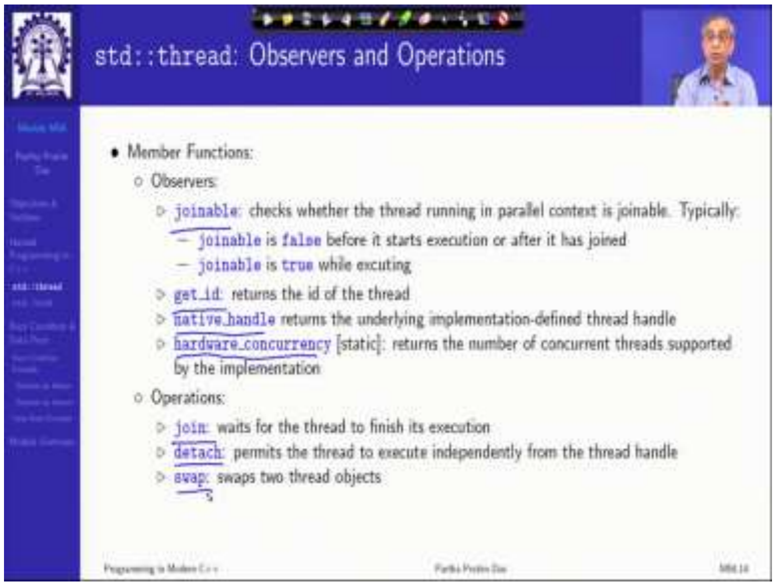
std::thread::~thread and std::thread::operato

- Member Functions:
  - `std::thread::~thread()`: Destructor: Destroys the thread object
    - If `*this` has an associated thread (`joinable() == true`), `std::terminate()` is called
    - A thread object does not have an associated thread (and is safe to destroy) after
      - it was default-constructed
      - it was moved from
      - `join()` has been called
      - `detach()` has been called
  - `std::thread::operator=`

    `thread& operator=(thread&& other) noexcept;`
    - If `*this` still has an associated running thread (that is, `joinable() == true`), call `std::terminate()`
    - Otherwise, assigns the state of other to `*this` and sets other to a default constructed state
    - After this call, `this->get_id()` is equal to the value of `other.get_id()` prior to the call, and other no longer represents a thread of execution

It has a destructor which, I mean which is executed at the end of its scope. There is an assignment operator, which basically is move assignment. So, if you, if you move a thread from a thread object to another, then this thread object will not contain any thread. Now, we will see this through examples.

(Refer Slide Time: 16:07)

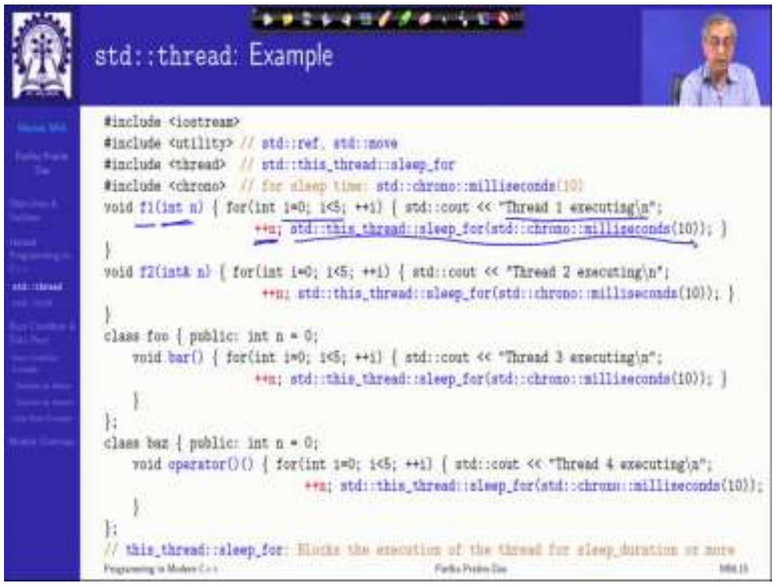


std::thread: Observers and Operations

- Member Functions:
  - Observers:
    - `joinable`: checks whether the thread running in parallel context is joinable. Typically:
      - `joinable` is `false` before it starts execution or after it has joined
      - `joinable` is `true` while excuting
    - `get_id`: returns the id of the thread
    - `native_handle` returns the underlying implementation-defined thread handle
    - `hardware_concurrency` [static]: returns the number of concurrent threads supported by the implementation
  - Operations:
    - `join`: waits for the thread to finish its execution
    - `detach`: permits the thread to execute independently from the thread handle
    - `swap`: swaps two thread objects

So, I am just leaving with pointers to you. There are certain observer functions like joinable. joinable is, is false before a thread starts execution or after it has joined in between while it is in execution, then it is joinable. Every thread has a unique ID which can get that. It has a native handle, you do not need not worry about these native handles and hardware concurrency right now. But the main thing is you can join, detach or swap thread objects.

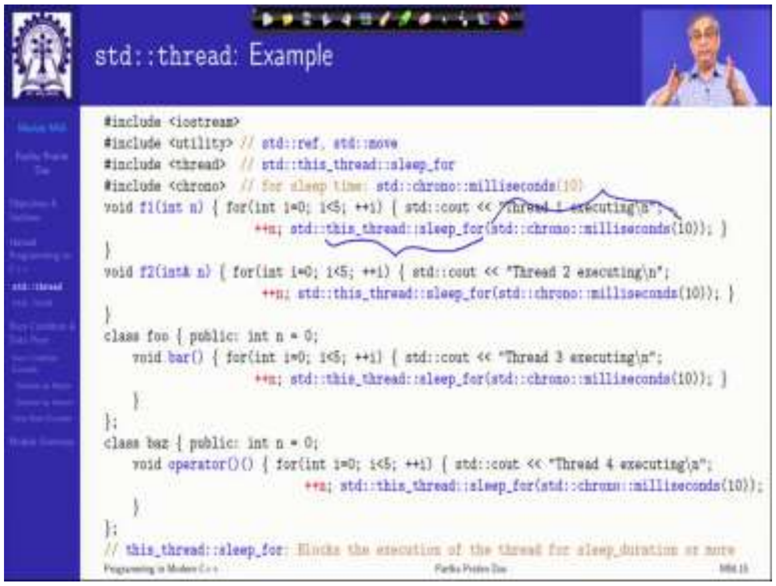
(Refer Slide Time: 16:42)



std::thread: Example

```
#include <iostream>
#include <utility> // std::ref, std::move
#include <thread>  // std::this_thread::sleep_for
#include <chrono>  // for sleep time: std::chrono::milliseconds(10)
void f1(int n) { for(int i=0; i<5; ++i) { std::cout << "Thread 1 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
}
void f2(int& n) { for(int i=0; i<5; ++i) { std::cout << "Thread 2 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
}
class foo { public: int n = 0;
    void bar() { for(int i=0; i<5; ++i) { std::cout << "Thread 3 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
    }
};
class baz { public: int n = 0;
    void operator()() { for(int i=0; i<5; ++i) { std::cout << "Thread 4 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
};
// this_thread::sleep_for: Blocks the execution of the thread for sleep_duration or more
```

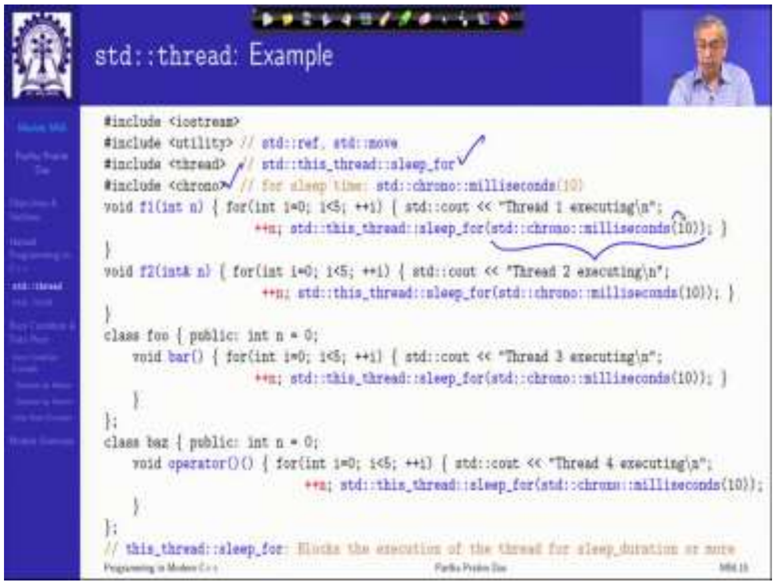
Programming in Modern C++ Partha Pratim Das M46.15



std::thread: Example

```
#include <iostream>
#include <utility> // std::ref, std::move
#include <thread>  // std::this_thread::sleep_for
#include <chrono>  // for sleep time: std::chrono::milliseconds(10)
void f1(int n) { for(int i=0; i<5; ++i) { std::cout << "Thread 1 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
}
void f2(int& n) { for(int i=0; i<5; ++i) { std::cout << "Thread 2 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
}
class foo { public: int n = 0;
    void bar() { for(int i=0; i<5; ++i) { std::cout << "Thread 3 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
    }
};
class baz { public: int n = 0;
    void operator()() { for(int i=0; i<5; ++i) { std::cout << "Thread 4 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
};
// this_thread::sleep_for: Blocks the execution of the thread for sleep_duration or more
```

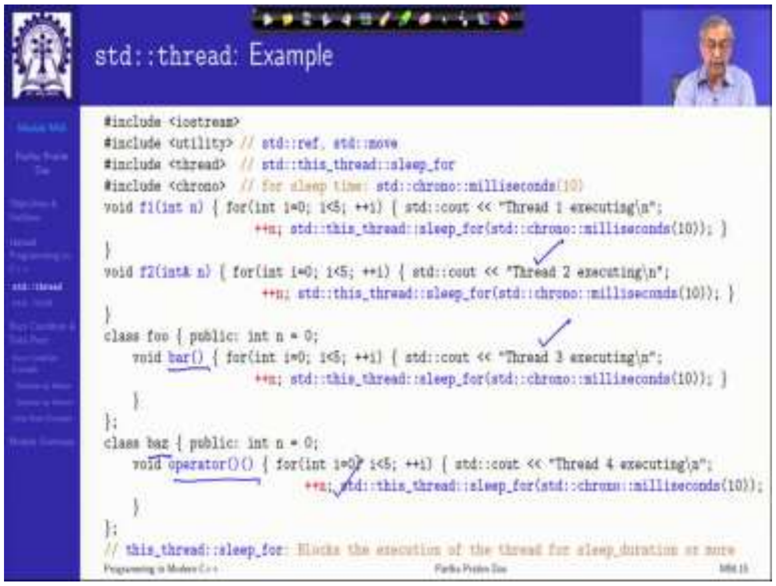
Programming in Modern C++ Partha Pratim Das M46.15



std::thread: Example

```
#include <iostream>
#include <utility> // std::ref, std::move
#include <thread>  // std::this_thread::sleep_for
#include <chrono>  // for sleep time: std::chrono::milliseconds(10)
void f1(int n) { for(int i=0; i<5; ++i) { std::cout << "Thread 1 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
}
void f2(int& n) { for(int i=0; i<5; ++i) { std::cout << "Thread 2 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
}
class foo { public: int n = 0;
    void bar() { for(int i=0; i<5; ++i) { std::cout << "Thread 3 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
    }
};
class baz { public: int n = 0;
    void operator()() { for(int i=0; i<5; ++i) { std::cout << "Thread 4 executing\n";
                            ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
};
// this_thread::sleep_for: Blocks the execution of the thread for sleep_duration or more
```

Programming in Modern C++ Partha Pratim Das M46.15



std::thread: Example

```
#include <iostream>
#include <utility> // std::ref, std::move
#include <thread>  // std::this_thread::sleep_for
#include <chrono>  // for sleep time: std::chrono::milliseconds(10)
void f1(int n) { for(int i=0; i<5; ++i) { std::cout << "Thread 1 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
}
void f2(int& n) { for(int i=0; i<5; ++i) { std::cout << "Thread 2 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
}
class foo { public: int n = 0;
    void bar() { for(int i=0; i<5; ++i) { std::cout << "Thread 3 executing\n";
                    ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10)); }
    }
};
class baz { public: int n = 0;
    void operator()() { for(int i=0; i<5; ++i) { std::cout << "Thread 4 executing\n";
                            ++n; std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
};
// this_thread::sleep_for: Blocks the execution of the thread for sleep_duration or more
```

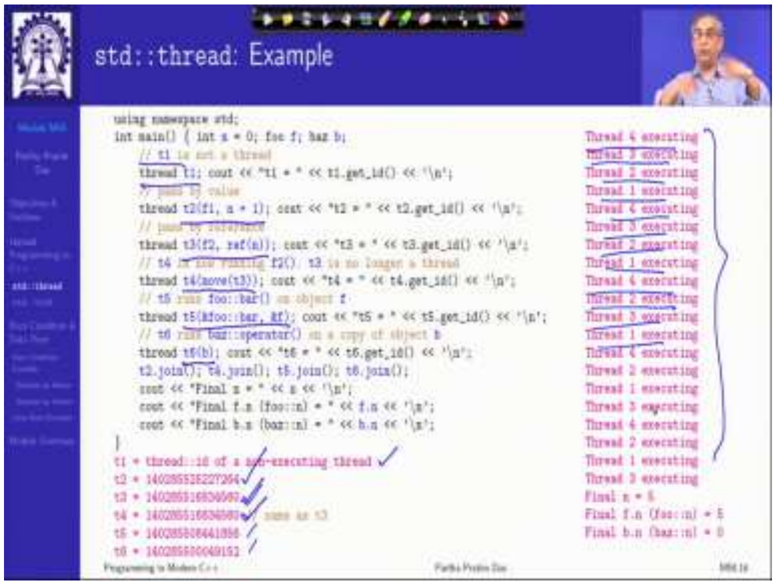
Programming in Modern C++ Partha Pratim Das M46.15

So, here is an example. So, let us look at this example I have a function f1 and which basically repeats four or five times writing thread one is executing. And then it increments a variable, the variable that it has got as a as a parameter, it increments that variable every time. Now then what it does is this long thing what it means is something simple. This is std::this_thread::sleep_for. So, what you are saying that whichever thread executes this is this thread.

And it says that this thread will now sleep, go to sleep for a certain amount of time. That is for this much time the thread will not do anything. So, that is what is available also in the thread component. And then you have to specify the time and that is available in the library in the chrono component when you say that chrono::millisecond that is nanosecond and so on also, and how much.

So, this click in just says that sleep for 10 milliseconds. The, the second function also does a similar thing with thread two, then you have a class which has a member function bar, which does it for thread three. Then there is there a another class baz, which is got the function operator overloaded it does the same thing and so on.

(Refer Slide Time: 18:27)



std::thread: Example

```
using namespace std;
int main() { int n = 0; foo f; baz b;
    // t1 is not a thread
    thread t1; cout << "t1 = " << t1.get_id() << '\n';
    // pass by value
    thread t2(f1, n + 1); cout << "t2 = " << t2.get_id() << '\n';
    // pass by reference
    thread t3(f2, ref(n)); cout << "t3 = " << t3.get_id() << '\n';
    // t4 is now running f2(). t3 is no longer a thread
    thread t4(move(t3)); cout << "t4 = " << t4.get_id() << '\n';
    // t5 runs foo::bar() on object f
    thread t5(&foo::bar, &f); cout << "t5 = " << t5.get_id() << '\n';
    // t6 runs baz::operator() on a copy of object b
    thread t6(b); cout << "t6 = " << t6.get_id() << '\n';
    t2.join(); t4.join(); t5.join(); t6.join();
    cout << "Final n = " << n << '\n';
    cout << "Final f.n (foo::n) = " << f.n << '\n';
    cout << "Final b.n (baz::n) = " << b.n << '\n';
}
```

```
t1 = thread::id of a non-executing thread ✓
t2 = 140285525227264 ✓
t3 = 140285516834560 ✓
t4 = 140285516834560 ✓ same as t3
t5 = 140285508441856 /
t6 = 140285500049152 /
```

```
Thread 4 executing
Thread 3 executing
Thread 2 executing
Thread 1 executing
Thread 4 executing
Thread 3 executing
Thread 2 executing
Thread 1 executing
Thread 4 executing
Thread 2 executing
Thread 3 executing
Thread 1 executing
Thread 4 executing
Thread 2 executing
Thread 1 executing
Thread 3 executing
Thread 4 executing
Thread 2 executing
Thread 1 executing
Thread 3 executing
Final n = 5
Final f.n (foo::n) = 5
Final b.n (baz::n) = 0
```

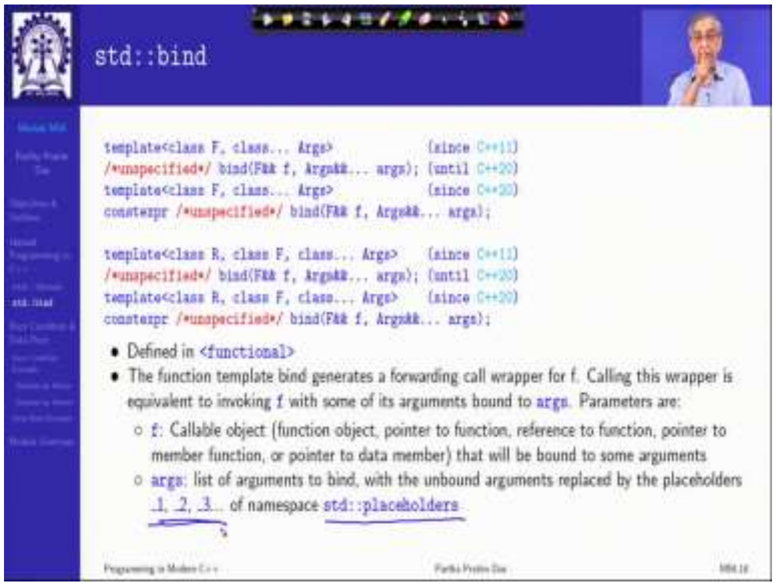
Programming in Modern C++ Partha Pratim Das

Now, if you execute them, so, you will see that here I have constructed a thread constructed a thread without giving it a function. So, it has a, it does not have an ID because it does not have any thread. I mean just the thread object default constructed. In the second we have given function f1 with a parameter n + 1. So, this is this is another way of constructing the thread. Earlier we did bind and then pass the functor, here we are directly you can directly pass that we saw that explicit very constructed with variadic templates.

So, it does construct t2. Similarly, we construct t3 with another, we construct t4 by moving t3. So, the thread from t3 goes to t4. Therefore, you will see that between t3 and t4 the trade IDs are same. Then I have t5 which is from the member function, I have t6 which is based on a on a on a functor object. So, all these will come and then they are executing and as they execute they keep on printing.

Now the difference is that as they execute, the order is not fixed. So, four threads are executing. But 4321, 4321 so they are not necessarily but but later on, if you just see that is the order, then we will see that it is 4231. So, you know, this order is indeterminable. So, this is basically the typical thread execution.

(Refer Slide Time: 20:25)



std::bind

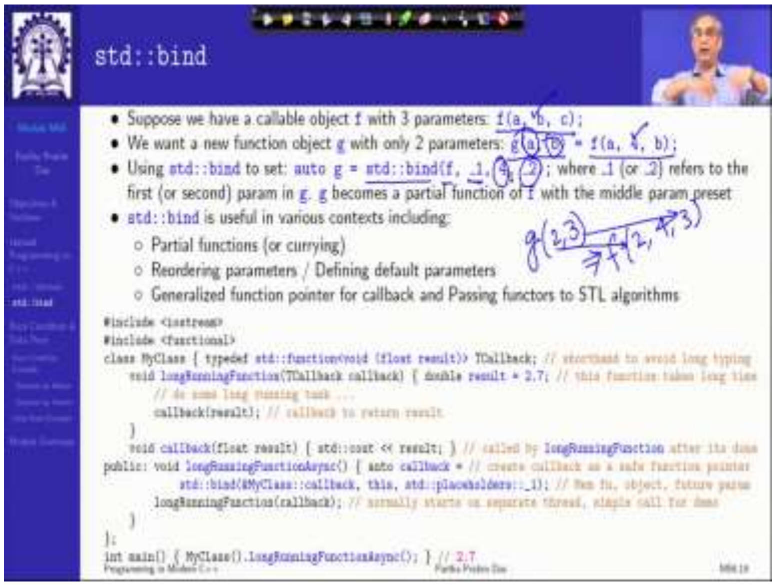
```
template<class F, class... Args>                 (since C++11)
/*unspecified*/ bind(F&& f, Args&&... args);  (until C++20)
template<class F, class... Args>                 (since C++20)
constexpr /*unspecified*/ bind(F&& f, Args&&... args);

template<class R, class F, class... Args>    (since C++11)
/*unspecified*/ bind(F&& f, Args&&... args);  (until C++20)
template<class R, class F, class... Args>    (since C++20)
constexpr /*unspecified*/ bind(F&& f, Args&&... args);
```

- Defined in `<functional>`
- The function template bind generates a forwarding call wrapper for f. Calling this wrapper is equivalent to invoking `f` with some of its arguments bound to `args`. Parameters are:
  - `f`: Callable object (function object, pointer to function, reference to function, pointer to member function, or pointer to data member) that will be bound to some arguments
  - `args`: list of arguments to bind, with the unbound arguments replaced by the placeholders _1, _2, _3... of namespace `std::placeholders`

Programming in Modern C++ Partha Pratim Das M84.18

Just a little word about bind, bind, as I said takes a function or a function called, any callable object and the parameters and put them together. And bind is a nice feature that it can put these parameters in the functor. And while putting them it can specify some of the parameters or leave some of the parameters for future. And there is a, there is a particular namespace called placeholder where these names, these are variable names, as you can see, underscore is a valid beginning of a variable name, identifier name, so underscore 1, 2, 3, et cetera, are defined.

(Refer Slide Time: 21:03)



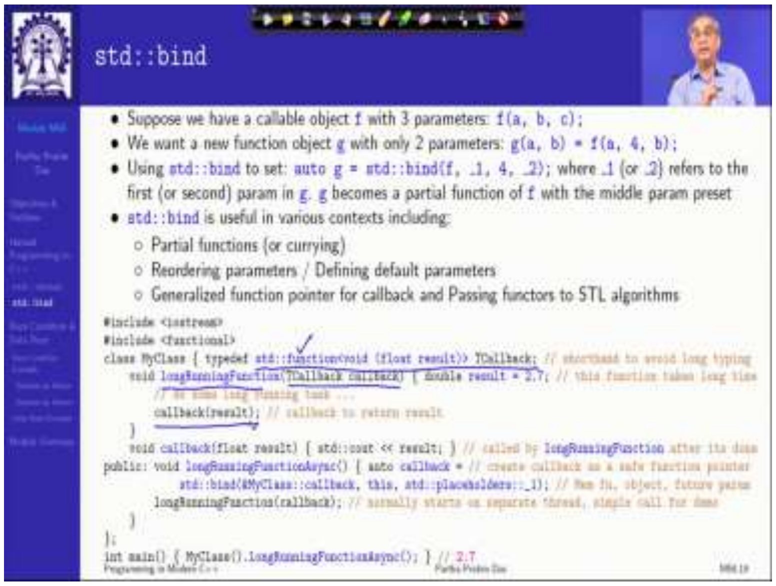
std::bind

- Suppose we have a callable object `f` with 3 parameters: `f(a, b, c);`
- We want a new function object `g` with only 2 parameters: `g(a, b) = f(a, 4, b);`
- Using `std::bind` to set: `auto g = std::bind(f, _1, 4, _2);` where `_1` (or `_2`) refers to the first (or second) param in `g`. `g` becomes a partial function of `f` with the middle param preset
- `std::bind` is useful in various contexts including:
  - Partial functions (or currying)
  - Reordering parameters / Defining default parameters
  - Generalized function pointer for callback and Passing functors to STL algorithms

$g(2,3) \Rightarrow f(2,4,3)$

```
#include <iostream>
#include <functional>
class MyClass { typedef std::function<void (float result)> TCallback; // shorthand to avoid long typing
    void longRunningFunction(TCallback callback) { double result = 2.7; // this function takes long time
        // do some long running task ...
        callback(result); // callback to return result
    }
    void callback(float result) { std::cout << result; } // called by longRunningFunction after its done
public: void longRunningFunctionAsync() { auto callback = // create callback as a safe function pointer
            std::bind(&MyClass::callback, this, std::placeholders::_1); // Mem fn, object, future param
        longRunningFunction(callback); // normally starts on separate thread, simple call for demo
    }
};
int main() { MyClass().longRunningFunctionAsync(); } // 2.7
```

Programming in Modern C++ Partha Pratim Das M44.19

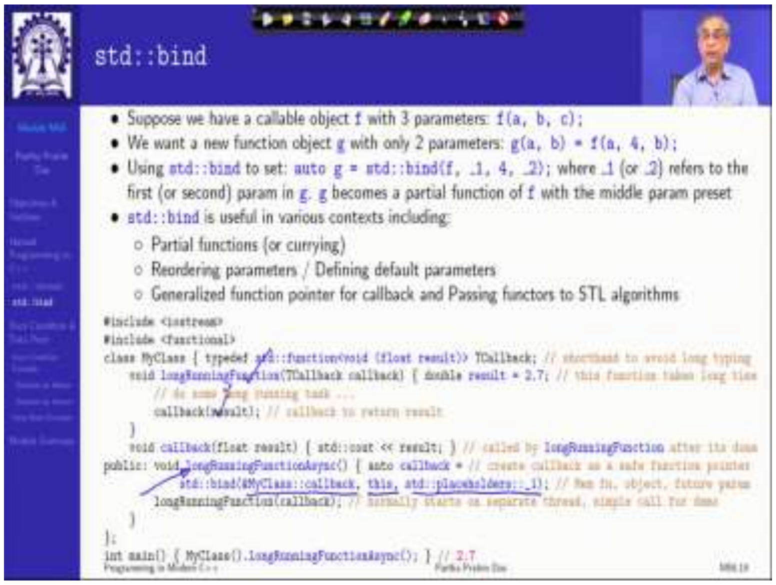


std::bind

- Suppose we have a callable object `f` with 3 parameters: `f(a, b, c);`
- We want a new function object `g` with only 2 parameters: `g(a, b) = f(a, 4, b);`
- Using `std::bind` to set: `auto g = std::bind(f, _1, 4, _2);` where `_1` (or `_2`) refers to the first (or second) param in `g`. `g` becomes a partial function of `f` with the middle param preset
- `std::bind` is useful in various contexts including:
  - Partial functions (or currying)
  - Reordering parameters / Defining default parameters
  - Generalized function pointer for callback and Passing functors to STL algorithms

```
#include <iostream>
#include <functional>
class MyClass { typedef std::function<void (float result)> TCallback; // shorthand to avoid long typing
    void longRunningFunction(TCallback callback) { double result = 2.7; // this function takes long time
        // do some long running task ...
        callback(result); // callback to return result
    }
    void callback(float result) { std::cout << result; } // called by longRunningFunction after its done
public: void longRunningFunctionAsync() { auto callback = // create callback as a safe function pointer
            std::bind(&MyClass::callback, this, std::placeholders::_1); // Mem fn, object, future param
        longRunningFunction(callback); // normally starts on separate thread, simple call for demo
    }
};
int main() { MyClass().longRunningFunctionAsync(); } // 2.7
```

Programming in Modern C++ Partha Pratim Das M44.19



std::bind

- Suppose we have a callable object f with 3 parameters: f(a, b, c);
- We want a new function object g with only 2 parameters: g(a, b) = f(a, 4, b);
- Using std::bind to set: auto g = std::bind(f, _1, 4, _2); where _1 (or _2) refers to the first (or second) param in g. g becomes a partial function of f with the middle param preset
- std::bind is useful in various contexts including:
  - Partial functions (or currying)
  - Reordering parameters / Defining default parameters
  - Generalized function pointer for callback and Passing functors to STL algorithms

```
#include <iostream>
#include <functional>
class MyClass { typedef std::function<void (float result)> TCallback; // shorthand to avoid long typing
    void longRunningFunction(TCallback callback) { double result = 2.7; // this function takes long time
        // do some long running task ...
        callback(result); // callback to return result
    }
    void callback(float result) { std::cout << result; } // called by longRunningFunction after its done
public: void longRunningFunctionAsync() { auto callback = // create callback as a safe function pointer
            std::bind(&MyClass::callback, this, std::placeholders::_1); // Mem fn, object, future param
        longRunningFunction(callback); // normally starts on separate thread, simple call for demo
    }
};
int main() { MyClass().longRunningFunctionAsync(); } // 2.7
```

Programming in Modern C++ Partha Pratim Das M66.19

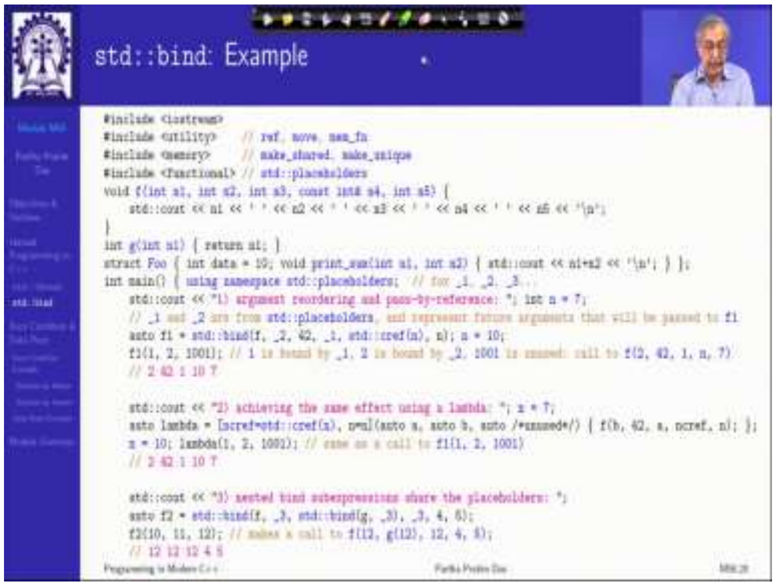
So, let us see, suppose you have a function with three parameters. And you want a new function with two parameters, where the middle parameter here should be four. So, you want this, you want. What you can do is obviously you can, you can write g and call this that is elaborate mechanism. But what you can do can do std::bind f. And then say _1, _1 will mean the first parameter of g which will be called.

Then the second parameter of f you keep as 4 fixed and the third parameter you passes _2, which means the second parameter of g in future. So, now when we will call g 2 3, it will actually mean a call, f 2, 4, 3. The first parameter comes here, and this order is arbitrary, you can put the first, I mean _2 first _1. So. you can basically shuffle around the order of the parameters, you can make certain parameters default, you can use it.

So, this bind basically makes it makes a very generalized to function pointer as a functor object. And it is very useful in passing parameters to STL algorithms. So, here below you have a, I have a simple example to show how it can be used as a as a callback. So, this is the callback type, you remember the function component. So, I say void float result. So, it means it will take a float, give me a void that is the type.

And I have a function that runs for a long time. So, what you do, you set another function call back function, which basically will be called while this function has completed its task. So, call back float result, which says this is the result. And that now, you wrap your long running function into an asynchronous function. You bind the call back with this particular function with a placeholder, which is _1, that is whatever you will give to call back as a parameter will go there. So, that is a result comes out here, just try this out. This is could fun.

(Refer Slide Time: 23:47)



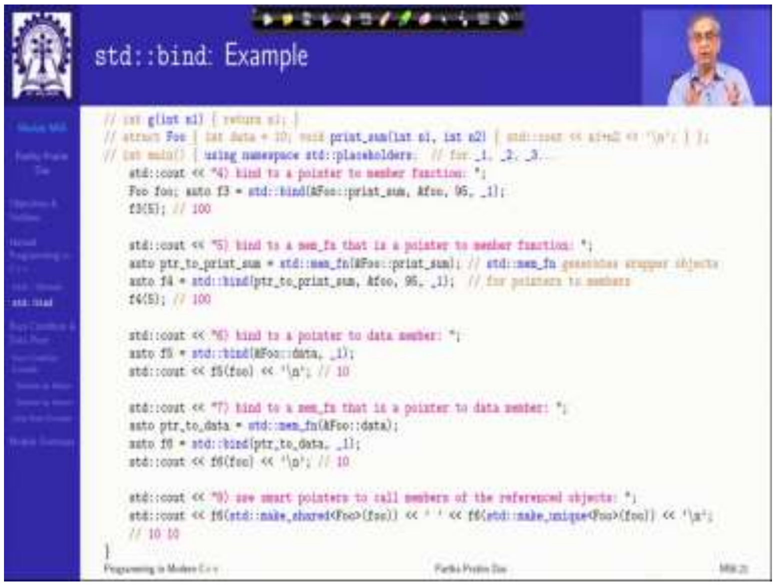
std::bind: Example

```
#include <iostream>
#include <utility>     // ref, move, mem_fn
#include <memory>      // make_shared, make_unique
#include <functional> // std::placeholders
void f(int n1, int n2, int n3, const int& n4, int n5) {
    std::cout << n1 << ' ' << n2 << ' ' << n3 << ' ' << n4 << ' ' << n5 << '\n';
}
int g(int n1) { return n1; }
struct Foo { int data = 10; void print_sum(int n1, int n2) { std::cout << n1+n2 << '\n'; } };
int main() { using namespace std::placeholders;  // for _1, _2, _3...
    std::cout << "1) argument reordering and pass-by-reference: "; int n = 7;
    // _1 and _2 are from std::placeholders, and represent future arguments that will be passed to f1
    auto f1 = std::bind(f, _2, 42, _1, std::cref(n), n); n = 10;
    f1(1, 2, 1001); // 1 is bound by _1, 2 is bound by _2, 1001 is unused; call to f(2, 42, 1, n, 7)
    // 2 42 1 10 7

    std::cout << "2) achieving the same effect using a lambda: "; n = 7;
    auto lambda = [ncref=std::cref(n), n=n](auto a, auto b, auto /*unused*/) { f(b, 42, a, ncref, n); };
    n = 10; lambda(1, 2, 1001); // same as a call to f1(1, 2, 1001)
    // 2 42 1 10 7

    std::cout << "3) nested bind subexpressions share the placeholders: ";
    auto f2 = std::bind(f, _3, std::bind(g, _3), _3, 4, 5);
    f2(10, 11, 12); // makes a call to f(12, g(12), 12, 4, 5);
    // 12 12 12 4 5
```

Programming in Modern C++ Partha Pratim Das M08.20

And in the next two slides, I have given different situations for bind, which you may just try out.

(Refer Slide Time: 23:51)



std::bind: Example

```
// int g(int n1) { return n1; }
// struct Foo { int data = 10; void print_sum(int n1, int n2) { std::cout << n1+n2 << '\n'; } };
// int main() { using namespace std::placeholders;  // for _1, _2, _3...
    std::cout << "4) bind to a pointer to member function: ";
    Foo foo; auto f3 = std::bind(&Foo::print_sum, &foo, 95, _1);
    f3(5); // 100

    std::cout << "5) bind to a mem_fn that is a pointer to member function: ";
    auto ptr_to_print_sum = std::mem_fn(&Foo::print_sum); // std::mem_fn generates wrapper objects
    auto f4 = std::bind(ptr_to_print_sum, &foo, 95, _1);  // for pointers to members
    f4(5); // 100

    std::cout << "6) bind to a pointer to data member: ";
    auto f5 = std::bind(&Foo::data, _1);
    std::cout << f5(foo) << '\n'; // 10

    std::cout << "7) bind to a mem_fn that is a pointer to data member: ";
    auto ptr_to_data = std::mem_fn(&Foo::data);
    auto f6 = std::bind(ptr_to_data, _1);
    std::cout << f6(foo) << '\n'; // 10

    std::cout << "8) use smart pointers to call members of the referenced objects: ";
    std::cout << f6(std::make_shared<Foo>(foo)) << ' ' << f6(std::make_unique<Foo>(foo)) << '\n';
    // 10 10
}
```

Programming in Modern C++ Partha Pratim Das M08.21

This is this is not code to thread programming, but I thought that bind is so widely used in terms of creating function objects for threads that you should be more familiar with it. So, I have given a detailed example for you to try.

(Refer Slide Time: 24:08)

Race Condition & Data Race

Race Condition & Data Race

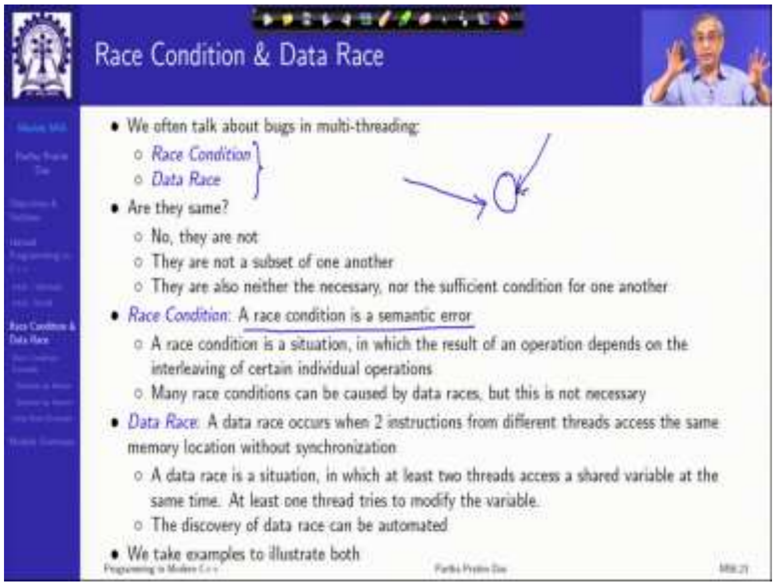
Sources:

- Concurrency in C++11, University of Chicago
- Race Conditions versus Data Races, 2017
- Malicious Race Conditions and Data Races, 2017
- C++11 Multithreading – Part 4: Data Sharing and Race Conditions
- C++11 Multithreading – Part 5: Using mutex to fix Race Conditions

Programming in Modern C++ Partha Pratim Das M08.22

Now, so that is the basics of, what happens in terms of threads.

(Refer Slide Time: 24:16)



Race Condition & 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

Programming in Modern C++ Partha Pratim Das M08.23

But threads are a big boon in terms of it, enhances performance, it allows you to, keep on when something is being done, and the system is busy with that you can still do something else in a different thread and so on, but they do not come without a price. So, there are different problems with threads. Now, the main is race condition and data race. So, these are talked about together. So you might have thought that they are the same thing.

They are related, but they are not the same thing, not at all. A race condition is more like a semantic error. A race condition is, is a situation where operations of multiple threads are kind of mixed up interleaved, so that they are not producing the right result. That is a race condition. And the data race is, when there is one single object, there are at least two threads, which are accessing them.

And at least one of them is trying to write. If multiple threads read a value, there is no problem. But if two or more threads try to read, write a value, at least one of them tries to write others maybe reading, then you have a data race. Often data leads to race condition. But data race, but test race condition may just happen by itself.

(Refer Slide Time: 25:37)

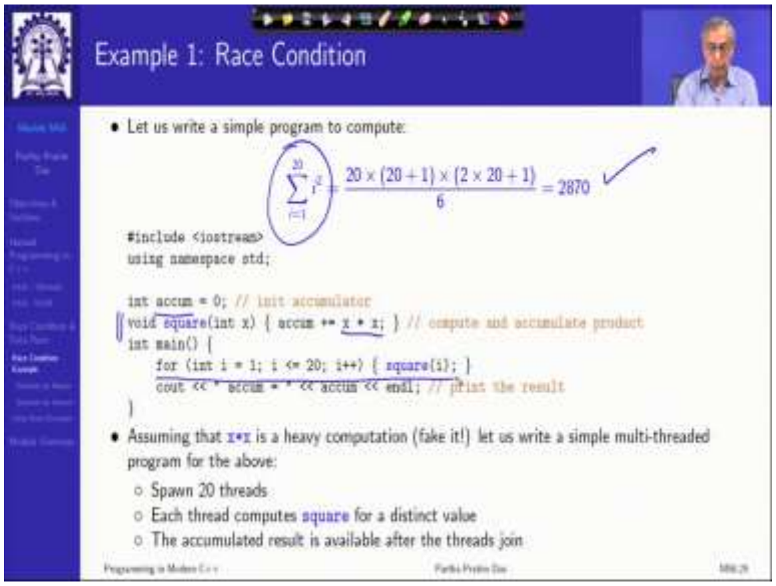
Race Condition & Data Race: Race Condition Example

Race Condition & Data Race: Race Condition Example

Programming in Modern C++ Partha Pratim Das M08.28

So, let us take a race condition example.

(Refer Slide Time: 25:40)



Example 1: Race Condition

- Let us write a simple program to compute:

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

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

int accum = 0; // init accumulator
void square(int x) { accum += x * x; } // compute and accumulate product
int main() {
    for (int i = 1; i <= 20; i++) { square(i); }
    cout << " accum = " << accum << endl; // print the result
}
```

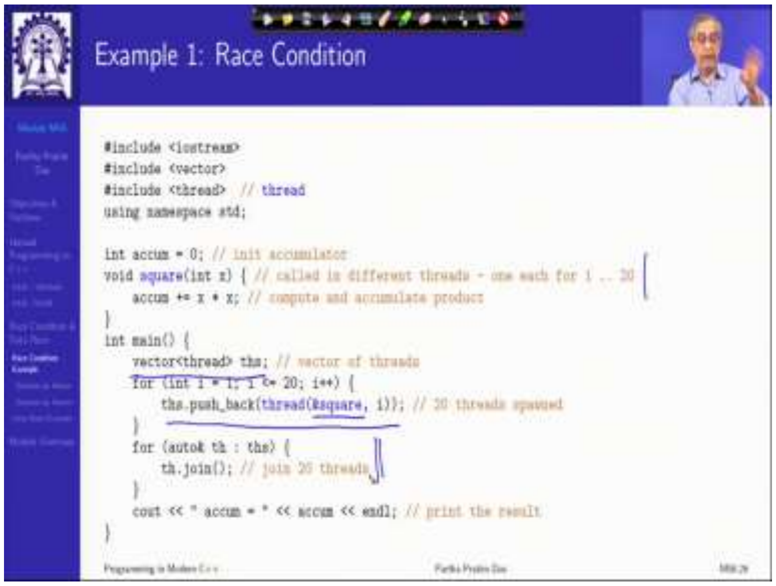
- Assuming that x*x is a heavy computation (fake it!) let us write a simple multi-threaded program for the above:
  - Spawn 20 threads
  - Each thread computes square for a distinct value
  - The accumulated result is available after the threads join

Programming in Modern C++ Partha Pratim Das M46.29

Suppose we want to do something very trivial, we want to work out the sum of squares, from 1 to 20. You know, this by this formula, this is what should happen if 1 square plus 2 square like that. And how we do that? We have a function square, which squares the parameter and updates a global accumulator. So, if I call square in a loop, one after the other, then the sum will get computed. This is a sequential program.

Now, let us try to do this, let us, let us just for the sake of it, assume that x into x, let it be a very heavy computation. So, I want to do them concurrently, when when 1 squared is being done, 2 square should be done concurrently, 3 squared should be done concurrently, and so on. They I mean x into x is not certainly heavy, but I am just using as a representative. So, what we will do, we will spawn 20 threads for 20 numbers, each thread will do the square for as much time it takes and update the accumulator. So, at the end, I should get the result.

(Refer Slide Time: 26:52)



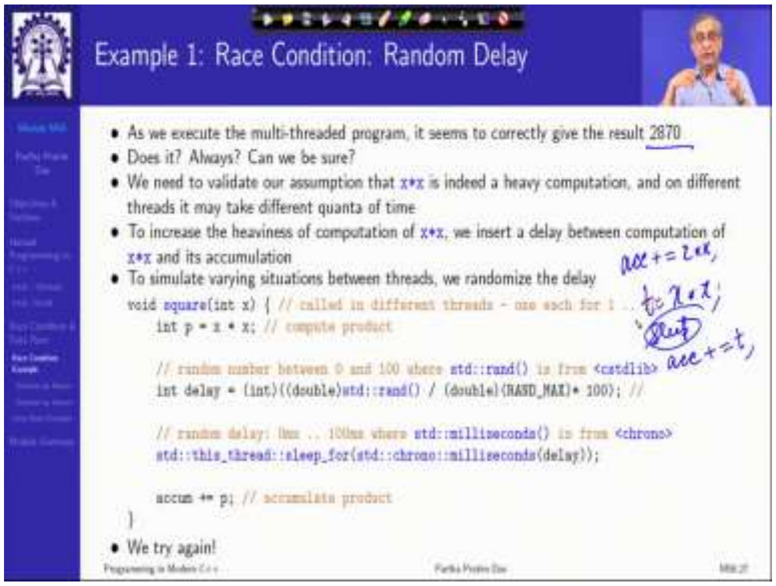
Example 1: Race Condition

```
#include <iostream>
#include <vector>
#include <thread>  // thread
using namespace std;

int accum = 0; // init accumulator
void square(int x) { // called in different threads - one each for 1 .. 20
    accum += x * x; // compute and 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 the result
}
```

Programming in Modern C++ Partha Pratim Das M08.26

Now, if I do that, so I have the thread version, there is no changes here. But here, what I do, I make a vector of threads. And I push the 20 threads, each with a different value of i for the function square. And I am just using a raw function pointer here. And I just put that in the vector, create and push them. So, they are all in the vector. As soon as they are created, they start working. And then I run over that vector to see if they have joined. So, this for loop will end when all of these threads have joined. And once that is done, my result is ready, very simple program.

(Refer Slide Time: 27:40)



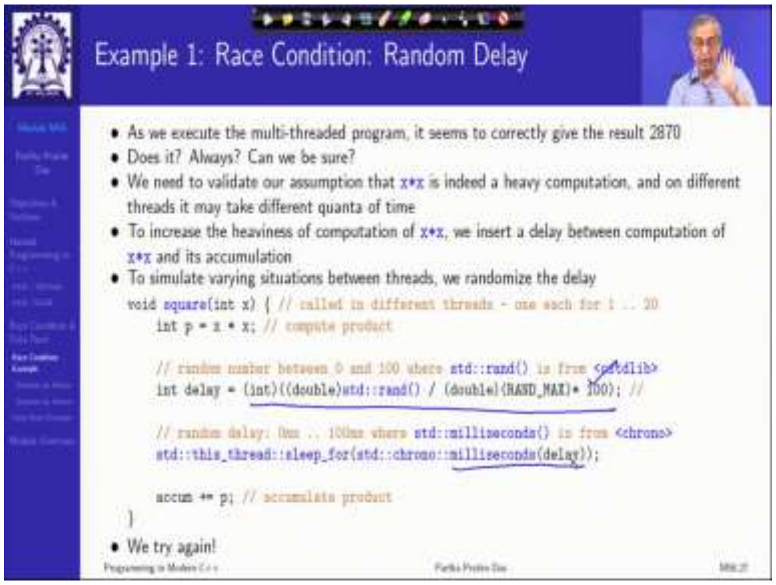
Example 1: Race Condition: Random Delay

- As we execute the multi-threaded program, it seems to correctly give the result 2870
- Does it? Always? Can we be sure?
- We need to validate our assumption that x*x is indeed a heavy computation, and on different threads it may take different quanta of time
- To increase the heaviness of computation of x*x, we insert a delay between computation of x*x and its accumulation
- To simulate varying situations between threads, we randomize the delay

```
void square(int x) { // called in different threads - one each for 1 ..
    int p = x * x; // compute product

    // random number between 0 and 100 where std::rand() is from <cstdlib>
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); //

    // random delay: 0ms .. 100ms where std::milliseconds() is from <chrono>
    std::this_thread::sleep_for(std::chrono::milliseconds(delay));

    accum += p; // accumulate product
}
```

- We try again!

Programming in Modern C++ Partha Pratim Das M08.27



Example 1: Race Condition: Random Delay

- As we execute the multi-threaded program, it seems to correctly give the result 2870
- Does it? Always? Can we be sure?
- We need to validate our assumption that `x*x` is indeed a heavy computation, and on different threads it may take different quanta of time
- To increase the heaviness of computation of `x*x`, we insert a delay between computation of `x*x` and its accumulation
- To simulate varying situations between threads, we randomize the delay

```
void square(int x) { // called in different threads - one each for 1 .. 20
    int p = x * x; // compute product

    // random number between 0 and 100 where std::rand() is from <cstdlib>
    int delay = (int)((double)std::rand() / (double)(RAND_MAX)* 100); //

    // random delay: 0ms .. 100ms where std::milliseconds() is from <chrono>
    std::this_thread::sleep_for(std::chrono::milliseconds(delay));

    accum += p; // accumulate product
}
```

- We try again!

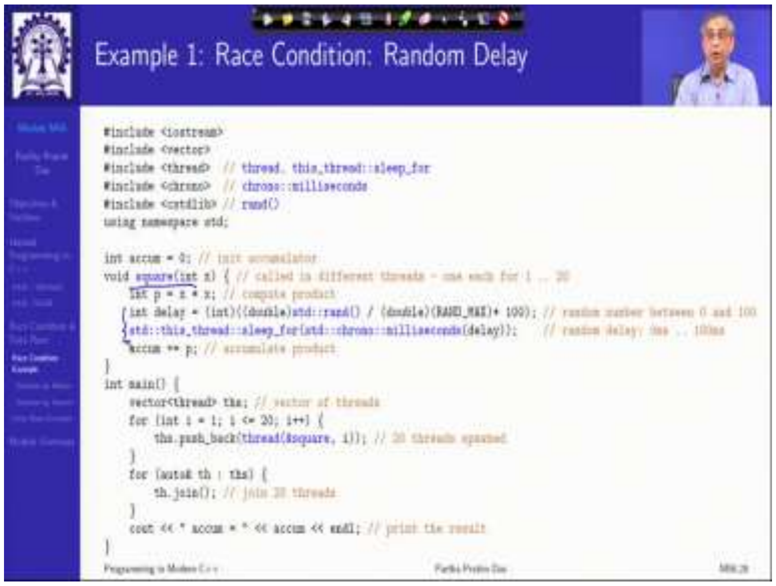
Programming in Modern C++ Partha Pratim Das M66.27

So, now, will it work? The question is, will it work? So, you try it, you try it and it gives the result 2840, 2870 at is suit. You keep on trying it and well, you always get the result. And you are still thinking that well, is it, is it correct? Is it correct? Will this work? So, let us take two things. One is that well, we had assumed that x into x is heavy, but we know that it is actually not heavy.

So, let us make it heavier. So, what we can do, instead of doing accumulator plus equal to x into x, let us compute x into x into some temporary. Let us sleep for a while and then add this temporary to the accumulator. This sleep effectively increases the perceived load or perceived computation time for x into x. So, that that way, I just make the load heavier by introducing a delay.

And then I say that it will not be the same for all numbers, 1 into 1 and 7 into 7 may not need the same amount of time. So, I just randomize, I put a random delay. So, what I do is I create a, I create, I generate a random number between 0 to RAND_MAX and normalize it to 100. So, that I get a random number between 0 to 100. And then I use that in the millisecond. So, some occasion it will be 0 millisecond, on 1 millisecond or 100 milliseconds. And we try it again.

(Refer Slide Time: 29:28)



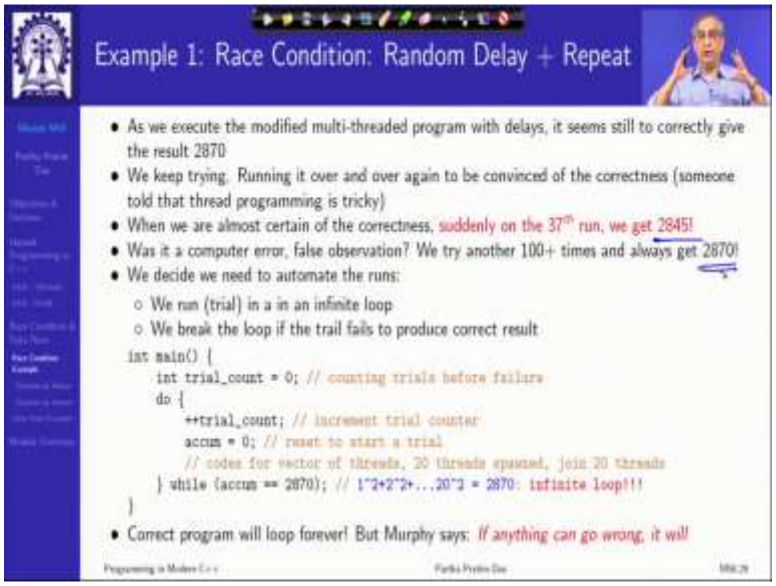
Example 1: Race Condition: Random Delay

```
#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() {
    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 the result
}
```

Programming in Modern C++ Partha Pratim Das M66.28

Now, so this is this is the total program, I am just so this is what you have done to put that random delay, which is what we have just discussed.

(Refer Slide Time: 29:39)



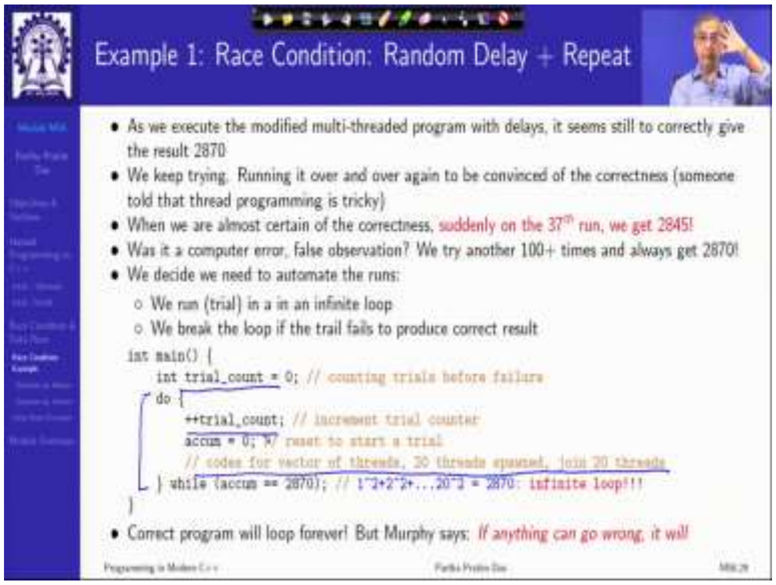
Example 1: Race Condition: Random Delay + Repeat

- As we execute the modified multi-threaded program with delays, it seems still to correctly give the result 2870
- We keep trying. Running it over and over again to be convinced of the correctness (someone told that thread programming is tricky)
- When we are almost certain of the correctness, suddenly on the 37th run, we get 2845!
- Was it a computer error, false observation? We try another 100+ times and always get 2870!
- We decide we need to automate the runs:
  - We run (trial) in a in an infinite loop
  - We break the loop if the trail fails to produce correct result

```
int main() {
    int trial_count = 0; // counting trials before failure
    do {
        ++trial_count; // increment trial counter
        accum = 0; // reset to start a trial
        // codes for vector of threads, 20 threads spawned, join 20 threads
    } while (accum == 2870); // 1^2+2^2+...20^2 = 2870: infinite loop!!!
}
```

- Correct program will loop forever! But Murphy says: *If anything can go wrong, it will*

Programming in Modern C++ Partha Pratim Das M66.29



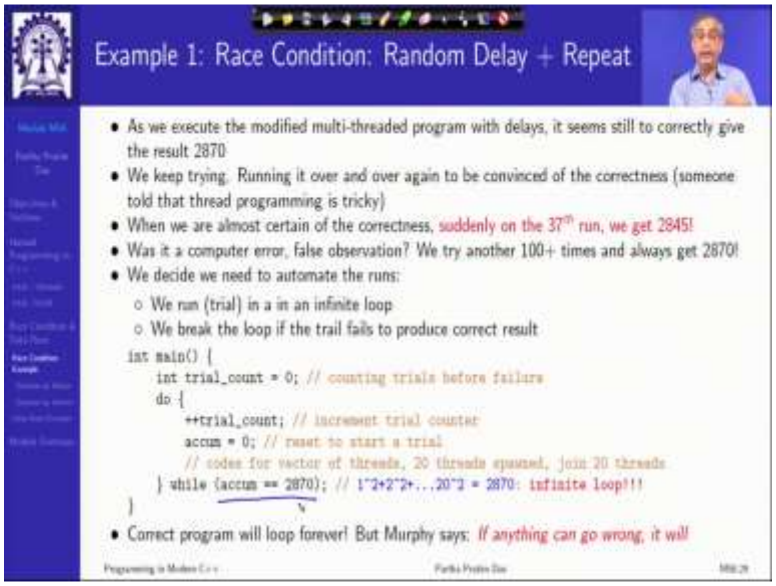
Example 1: Race Condition: Random Delay + Repeat

- As we execute the modified multi-threaded program with delays, it seems still to correctly give the result 2870
- We keep trying. Running it over and over again to be convinced of the correctness (someone told that thread programming is tricky)
- When we are almost certain of the correctness, suddenly on the 37th run, we get 2845!
- Was it a computer error, false observation? We try another 100+ times and always get 2870!
- We decide we need to automate the runs:
  - We run (trial) in a in an infinite loop
  - We break the loop if the trail fails to produce correct result

```
int main() {
    int trial_count = 0; // counting trials before failure
    do {
        ++trial_count; // increment trial counter
        accum = 0; // reset to start a trial
        // codes for vector of threads, 20 threads spawned, join 20 threads
    } while (accum == 2870); // 1^2+2^2+...20^2 = 2870: infinite loop!!!
}
```

- Correct program will loop forever! But Murphy says: *If anything can go wrong, it will*

Programming in Modern C++ Partha Pratim Das M66.29



Example 1: Race Condition: Random Delay + Repeat

- As we execute the modified multi-threaded program with delays, it seems still to correctly give the result 2870
- We keep trying. Running it over and over again to be convinced of the correctness (someone told that thread programming is tricky)
- When we are almost certain of the correctness, suddenly on the 37th run, we get 2845!
- Was it a computer error, false observation? We try another 100+ times and always get 2870!
- We decide we need to automate the runs:
  - We run (trial) in a in an infinite loop
  - We break the loop if the trail fails to produce correct result

```
int main() {
    int trial_count = 0; // counting trials before failure
    do {
        ++trial_count; // increment trial counter
        accum = 0; // reset to start a trial
        // codes for vector of threads, 20 threads spawned, join 20 threads
    } while (accum == 2870); // 1^2+2^2+...20^2 = 2870: infinite loop!!!
}
```

- Correct program will loop forever! But Murphy says: *If anything can go wrong, it will*

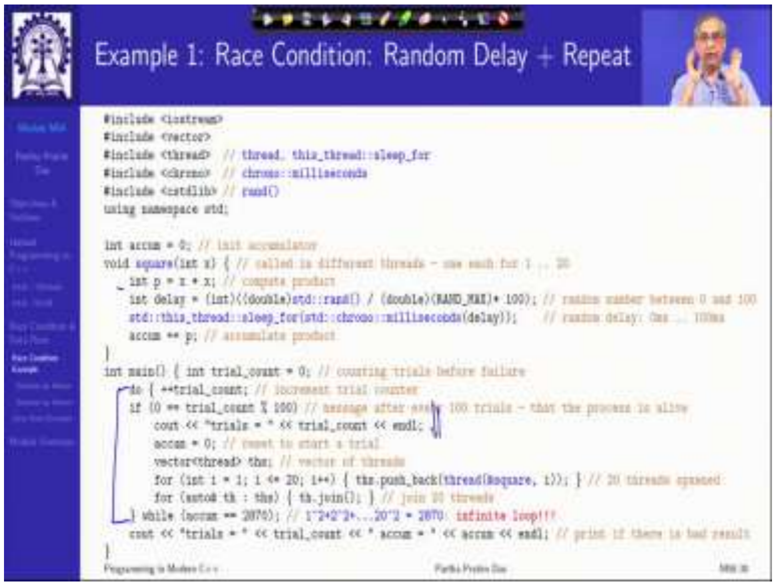
Programming in Modern C++ Partha Pratim Das M66.29

Now, we say that, we again keep trying, it gives the correct result. I keep trying, keep trying, I did this for 36 times I got to 2870. But on the 37th attempt, suddenly I got a result 2845. The question is, what is the problem? Is it, is it a computer error or a false observation? So, I tried 100 more times, more than 100 times, but I always got to 2870. So, I realized that well, before I can be convinced, I need to have some automated test for this.

I cannot just keep on every time, go to online gdb and run run run, run, getting mad. So, how can I do an automated test? It is very simple, I will just put this whole stuff of, creating the vector of threads, spawning the 20 threads and joining up the threads within a scope, so that they are all localized, and put that in a do, do while loop. When have a trial count variable, every time it goes through this loop, this whole run has happened once.

And that trial count is incremented. And how do I terminate the loop? I terminate the loop by checking if the accumulator is 2870. If that accumulator is 2870, it is correct. So, then I will try again. If it is not, then I will exit. But that means that if the program is correct, it will be an infinite loop. Well, but then Murphy's Law says that, if anything can go wrong, it will. So, let us see what we find.

(Refer Slide Time: 31:24)



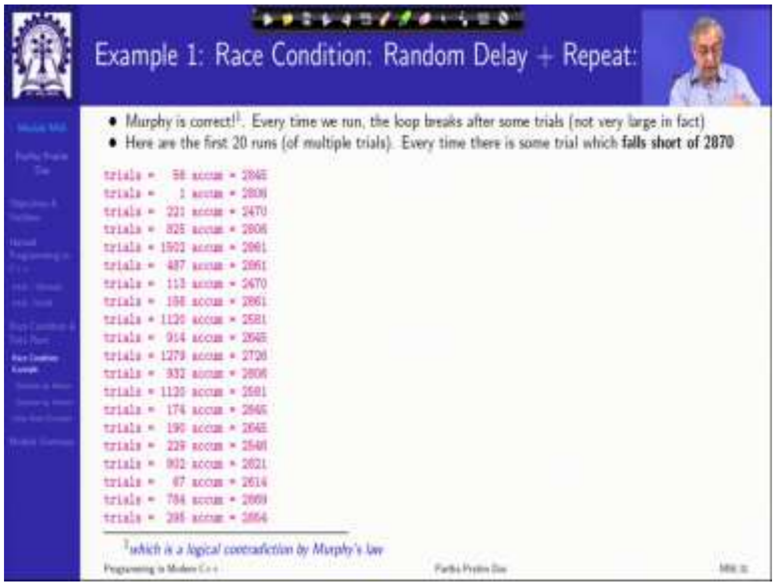
Example 1: Race Condition: Random Delay + Repeat

```
#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
}
```

Programming in Modern C++ Partha Pratim Das

So with this, this is, again, the entire program, there is a random delay, and this is a repeat. And since it might go on for a long time, what I do is every, after every 100 trials, I just print a message on the console, that 100 trials done, 200 trials done so that, you know, I know that the program is working, it is alive. And then I wait, if a wrong result would come.

(Refer Slide Time: 31:52)



Example 1: Race Condition: Random Delay + Repeat:

- Murphy is correct![1]. Every time we run, the loop breaks after some trials (not very large in fact)
- Here are the first 20 runs (of multiple trials). Every time there is some trial which **falls short of 2870**

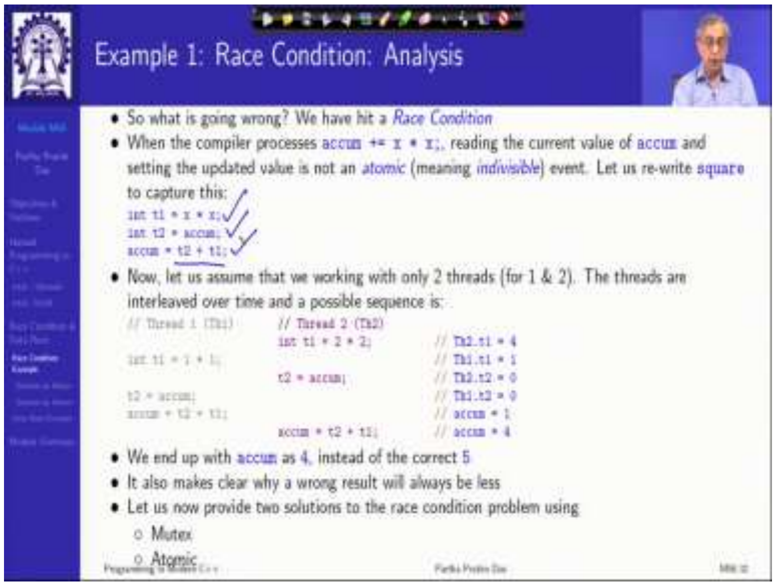
```
trials =   56 accum = 2845
trials =    1 accum = 2806
trials =  221 accum = 2470
trials =  825 accum = 2806
trials = 1502 accum = 2861
trials =  487 accum = 2861
trials =  113 accum = 2470
trials =  156 accum = 2861
trials = 1120 accum = 2581
trials =  914 accum = 2845
trials = 1279 accum = 2726
trials =  932 accum = 2806
trials = 1120 accum = 2581
trials =  174 accum = 2845
trials =  190 accum = 2845
trials =  229 accum = 2540
trials =  902 accum = 2821
trials =   67 accum = 2614
trials =  784 accum = 2869
trials =  295 accum = 2854
```

[1] *which is a logical contradiction by Murphy's law*

Programming in Modern C++ Partha Pratim Das M46.31

Murphy is correct. I did this for 20 times. And you can see this is the actual data I collected. So, when I run this program for the first time in the 56th trial, it produces 2845. In the second trial, second run, in the first trial itself, it produced 2806, and so on. In some cases it has taken second 1502 runs to produce a wrong result, but in every case, 20 cases it has eventually proved it. So, there is something wrong about this. And if you look at all these values, you will see that these values are less than 2870, all of them.

(Refer Slide Time: 32:38)



Example 1: Race Condition: Analysis

- So what is going wrong? We have hit a *Race Condition*
- When the compiler processes `accum += x * x;`, reading the current value of `accum` and setting the updated value is not an *atomic* (meaning *indivisible*) event. Let us re-write `square` to capture this:

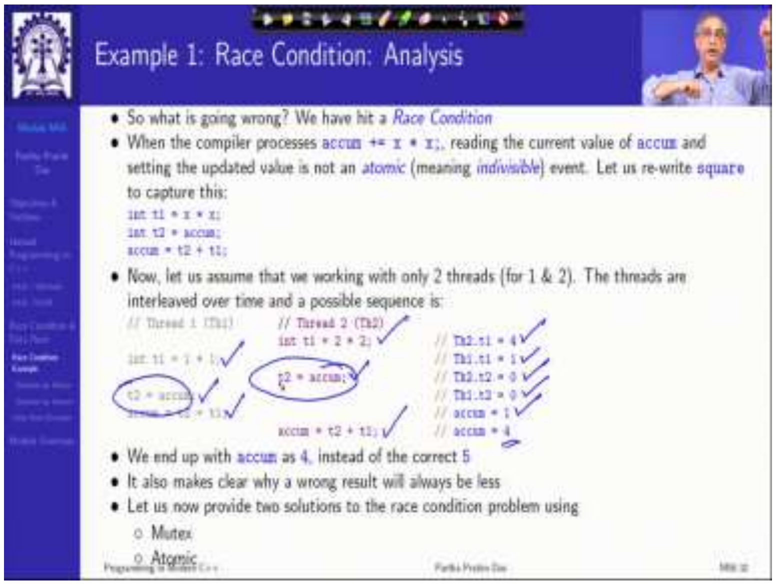
```
int t1 = x * x; ✓
int t2 = accum; ✓
accum = t2 + t1; ✓
```

- Now, let us assume that we working with only 2 threads (for 1 & 2). The threads are interleaved over time and a possible sequence is:

```
// Thread 1 (Th1)      // Thread 2 (Th2)
                       int t1 = 2 * 2;        // Th2.t1 = 4
int t1 = 1 * 1;                               // Th1.t1 = 1
                       t2 = accum;            // Th2.t2 = 0
t2 = accum;                                   // Th1.t2 = 0
accum = t2 + t1;                              // accum = 1
                       accum = t2 + t1;       // accum = 4
```

- We end up with `accum` as 4, instead of the correct 5
- It also makes clear why a wrong result will always be less
- Let us now provide two solutions to the race condition problem using
  - Mutex
  - Atomic

Programming in Modern C++ Partha Pratim Das M46.32



Example 1: Race Condition: Analysis

- So what is going wrong? We have hit a *Race Condition*
- When the compiler processes `accum += x * x;`, reading the current value of `accum` and setting the updated value is not an *atomic* (meaning *indivisible*) event. Let us re-write `square` to capture this:

```
int t1 = x * x;
int t2 = accum;
accum = t2 + t1;
```

- Now, let us assume that we working with only 2 threads (for 1 & 2). The threads are interleaved over time and a possible sequence is:

```
// Thread 1 (Th1)        // Thread 2 (Th2)
                         int t1 = 2 * 2;         // Th2.t1 = 4
int t1 = 1 * 1;                                  // Th1.t1 = 1
                         t2 = accum;             // Th2.t2 = 0
t2 = accum;                                      // Th1.t2 = 0
accum = t2 + t1;                                 // accum = 1
                         accum = t2 + t1;        // accum = 4
```

- We end up with `accum` as 4, instead of the correct 5
- It also makes clear why a wrong result will always be less
- Let us now provide two solutions to the race condition problem using
  - Mutex
  - Atomic

Programming in Modern C++ Partha Pratim Das

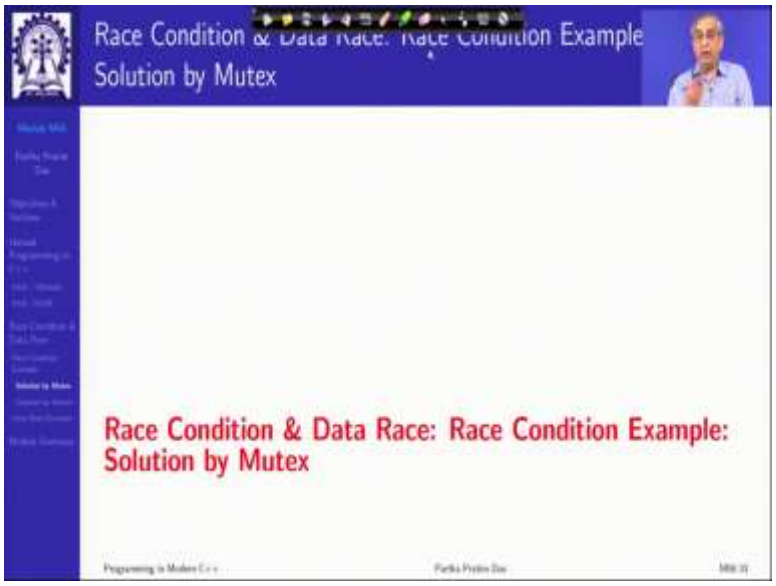
So, what is going on? So, to understand, let us you know, you know, split into the simple square function as to what it is doing. So, what will the compiler do? Compiler has to multiply. So, the code written is necessarily x * x. Now the compiler has to compute x turn x into a temporary has to read the accumulator into another temporary.

And then add these two temporaries. And put in the accumulator. This is a basic process. So, let us say I have two threads. And they are they are independent. So, they the order in which those instructions are executing, are can interleave in any way. But at any cycle only or at any point, the instruction of only one thread can execute because there is a, there is only one processor to execute.

So, let us say if they are interleaved in this way that it first does this in thread two, then it does this in thread one. So, in thread 2 t1 is 4, in thread 1 t1 is 1, then it does this. So, in thread 2, t2 is 0, because accumulator initially is 0. Then in thread 1, t2 becomes 0, because accumulator is 0. Then it updates the accumulator. So, in thread by thread one accumulator becomes 1 and then it does this, accumulator becomes 4.

But it should have been 1 + 4 5. So, you can see that the interleaving only because that both these threads have in this process, started with a value of the accumulator which is known updated one, and both of them have updated after that, because of this interleaving. And that is the sole problem of having the incorrect result. And you can always see that it will always have a lower result than what is expected. And that is what you have been seeing. And this is exactly what is called the race condition.

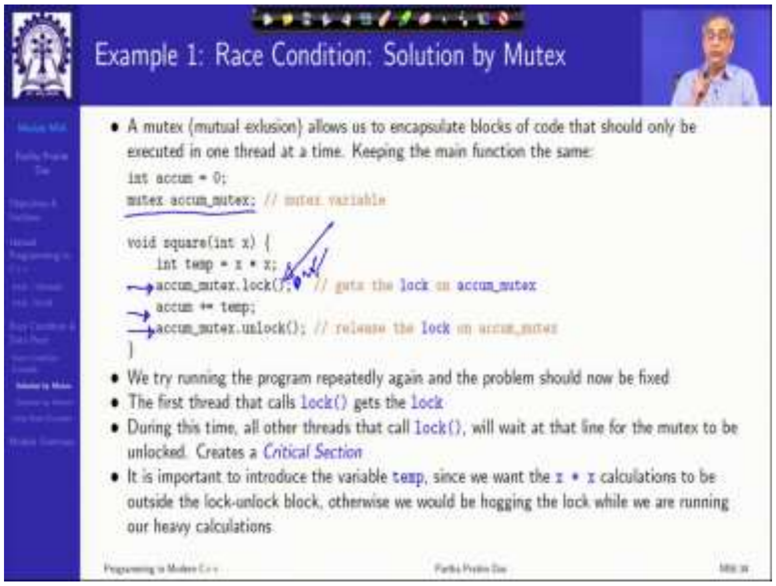
(Refer Slide Time: 34:47)



Race Condition & Data Race: Race Condition Example: Solution by Mutex

So, how to solve that? How to fix that? There are multiple solutions.

(Refer Slide Time: 34:51)



Example 1: Race Condition: Solution by Mutex

- A mutex (mutual exlusion) 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
}
```

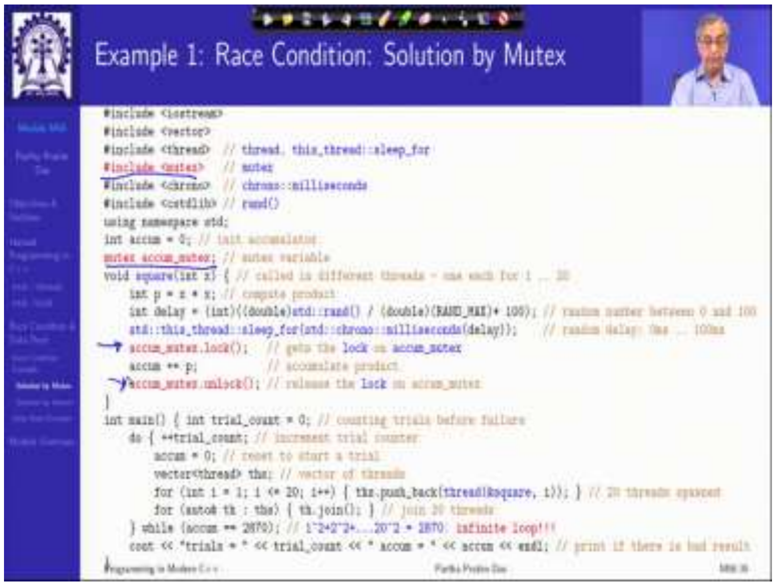
- We try running the program repeatedly again and the problem should now be fixed
- The first thread that calls lock() gets the lock
- During this time, all other threads that call lock(), will wait at that line for the mutex to be unlocked. Creates a *Critical Section*
- It is important to introduce 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

So, I will just talk about two. One is solution by mutex. There is something like mutex that is defined, which is in the mutex component. So, it is a mutex and give it a mutex variable name. Then you do a lock on the mutex. If you do a lock on the mutex, what it does, it is an atomic operation and what it does, the first thread that comes here will get the lock. And will be allowed to go and do the next instruction.

But the second thread which comes to this point, will not get the lock because the lock has been given to one thread. So, it is like having a unique lock, unique key. So, key, the thread which is having the lock has not released it. Any other thread, all other threads will keep on waiting at this point.

And when the thread which has the lock unlocks it, then one of these threads will get the lock again. And that will start proceeding others will wait. So, this will make sure that at any point of time, only one thread, will be able to read and write that monitor because this is both read and write of the accumulator.

(Refer Slide Time: 36:06)



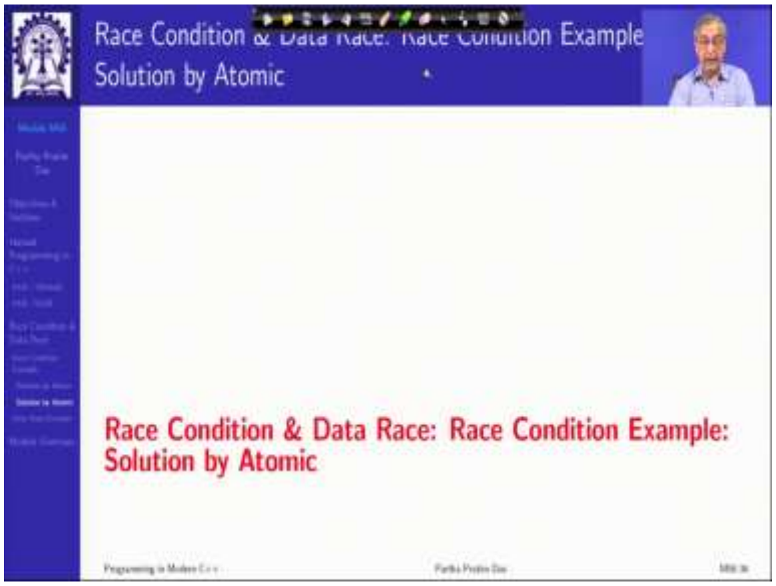
Example 1: Race Condition: Solution by Mutex

```
#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() { int trial_count = 0; // counting trials before failure
    do { ++trial_count; // increment trial counter
        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
```

Programming in Modern C++ Partha Pratim Das

And we put it in the solution very simply fully few lines I changed include this, have a mutex and put this locks and then try again. Like our, our repeated trials with delay, if we do that, if we try again and we will see.

(Refer Slide Time: 36:20)

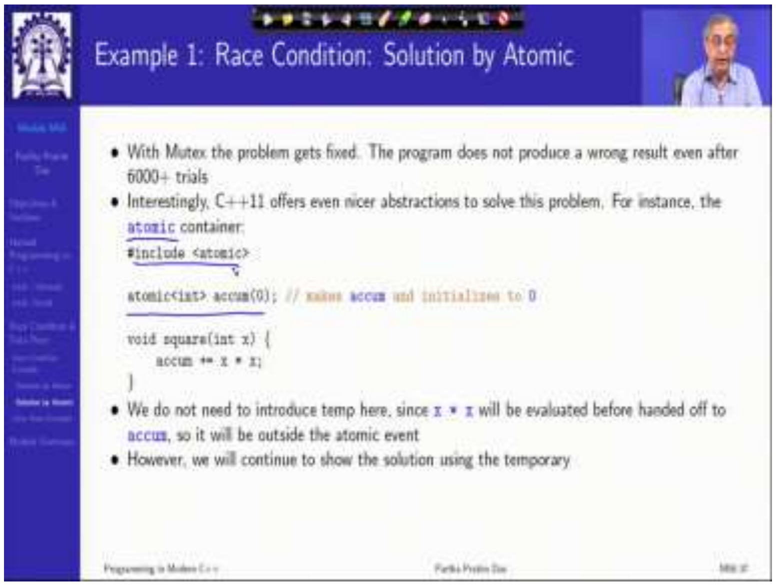


Race Condition & Data Race: Race Condition Example: Solution by Atomic

Race Condition & Data Race: Race Condition Example: Solution by Atomic

I have waited up to 6000 runs of this and the program did not terminate, this is truly an infinite loop.

(Refer Slide Time: 36:30)



Example 1: Race Condition: Solution by Atomic

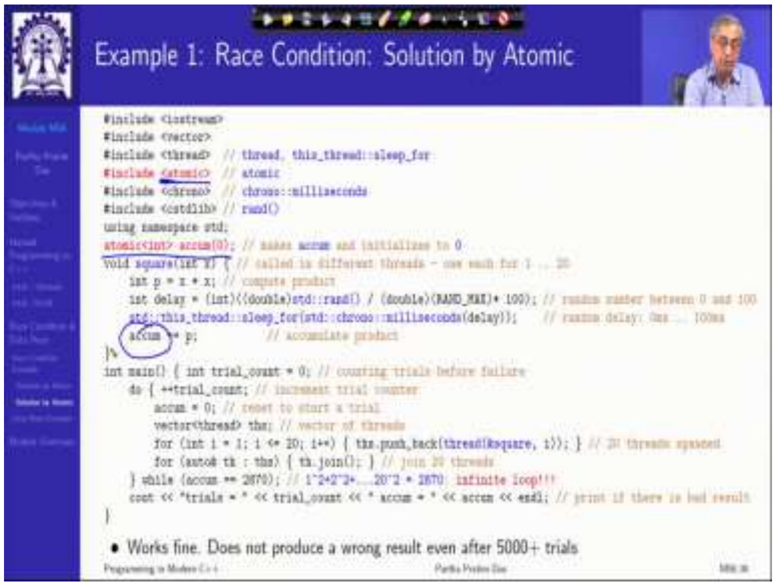
- With Mutex the problem gets fixed. The program does not produce a wrong result even after 6000+ trials
- Interestingly, C++11 offers even nicer abstractions to solve this problem. For instance, the atomic container:

```
#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

And other way of doing this is using what is known as atomic container. Atomic is a specific type of container, where whatever you put as an atomic container can be changed only by one thread, not by others. So, you can just say atomic include this say atomic and things will get done.

(Refer Slide Time: 37:00)



Example 1: Race Condition: Solution by Atomic

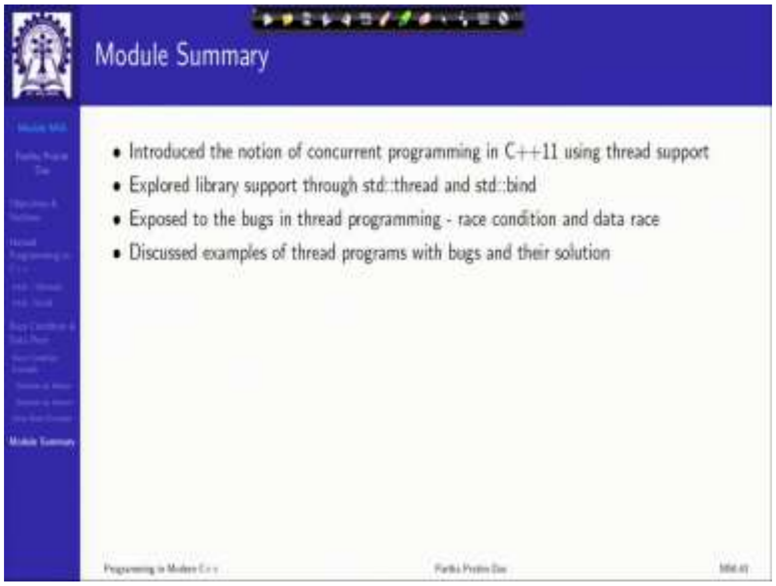
```
#include <iostream>
#include <vector>
#include <thread>  // thread, this_thread::sleep_for
#include <atomic>  // atomic
#include <chrono>  // chrono::milliseconds
#include <cstdlib> // rand()
using namespace std;
atomic<int> accum(0); // makes accum and initializes to 0
void square(int x) { // called 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
        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
}
```

- Works fine. Does not produce a wrong result even after 5000+ trials

Programming in Modern C++ Partha Pratim Das

And this is so just these two inclusion of this. And instead of a global declaration, you just say atomic int. So, it gives you an atomic integer so that when it is trying to change the accumulator, which is atomic, no other thread will be able to read or write into that. This also solves the problem thread. We tried it 15,000 plus times and everything was okay.

(Refer Slide Time: 37:31)



Module Summary

- 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

So, this is broadly the basic introduction to the thread programming and the race condition. We will talk more about these in the next module, but we have got a sense of how to create threads, how to run them how to join, how to bind functions, or functors to the thread and how to solve a very simple problems of race condition using mutex or atomic. We will discuss more of these in the next module. Thank you very much for your attention.